Verteilte Systeme
Vorstellung und Wiederholung

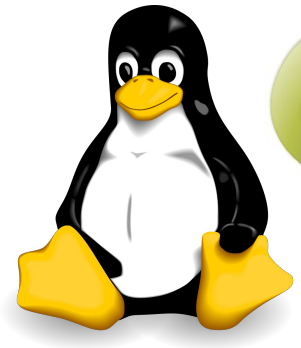
Der Schulmeister

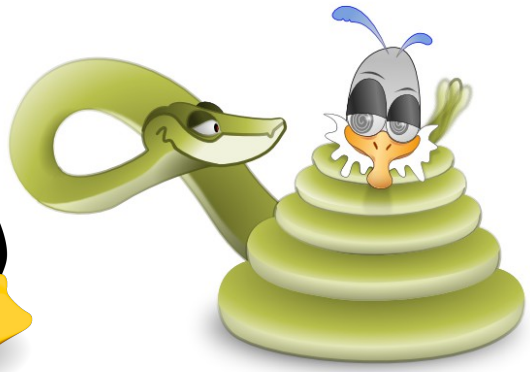
Dennis Schulmeister

Dipl.-Wirtschaftsinformatiker (BA)

Seniorentwickler SAP AddOn
Debitorenmanagement bei der
cormeta ag, Ettlingen

Mal wieder: Ihr Dozent





Was wir machen werden ...

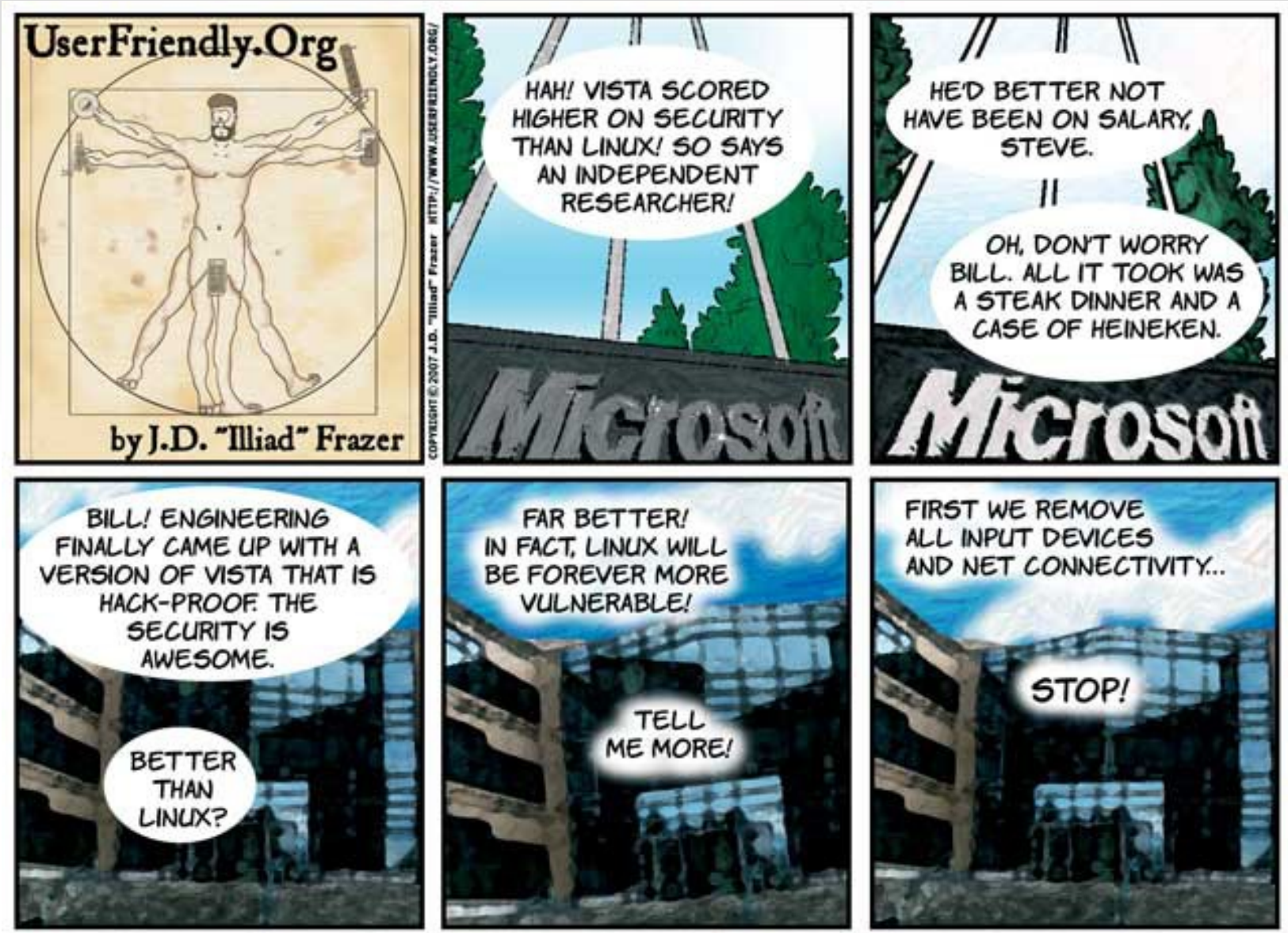
Anwendungen auf mehrere Rechner verteilen

- Client/Server-Anwendungen mit Sockets
- Entfernte Aufrufe mit WebServices und Middleware
- Asynchrone Aufrufe mit Nachrichtenwarteschlangen
- Große Anwendungen mit der Java Enterprise Edition

Komponentenbasierte Softwarearchtektur

Spiel, Spaß und Unterhaltung (Teamarbeit)

… und was nicht



Vorlesungsinhalte

Definition und Grundlagen verteilter Systeme

Entwurfsmuster und Komponentenmodelle

Von Sockets zur Middleware

- Wiederholung Socketprogrammieruing
- Kommunikationsorientierte Middleware
- Anwendungsorientierte Middleware

Entfernte Aufrufe

- Remote Procedure Call / Remote Method Invocation
- Aufruf und Erstellung von WebServices

Vorlesungsinhalte

Java Enterprise Edition

- Java Message Services (Asynchrone Nachrichten)
- Java Naming and Directory Services (Verzeichnisse)
- Enterprise Java Beans und Java Persistence API

Teamarbeit zwischen den Vorlesungen

- Vier bis fünf Gruppen mit je ca. fünf Teilnehmern
- Präsentation der Aufgaben alle ein bis zwei Wochen
- **Aufgaben:** Kleine Netzwerkspiele, Chatserver und Chatbots, Onlineanwendungen, Webservices, …

Was Sie schon können

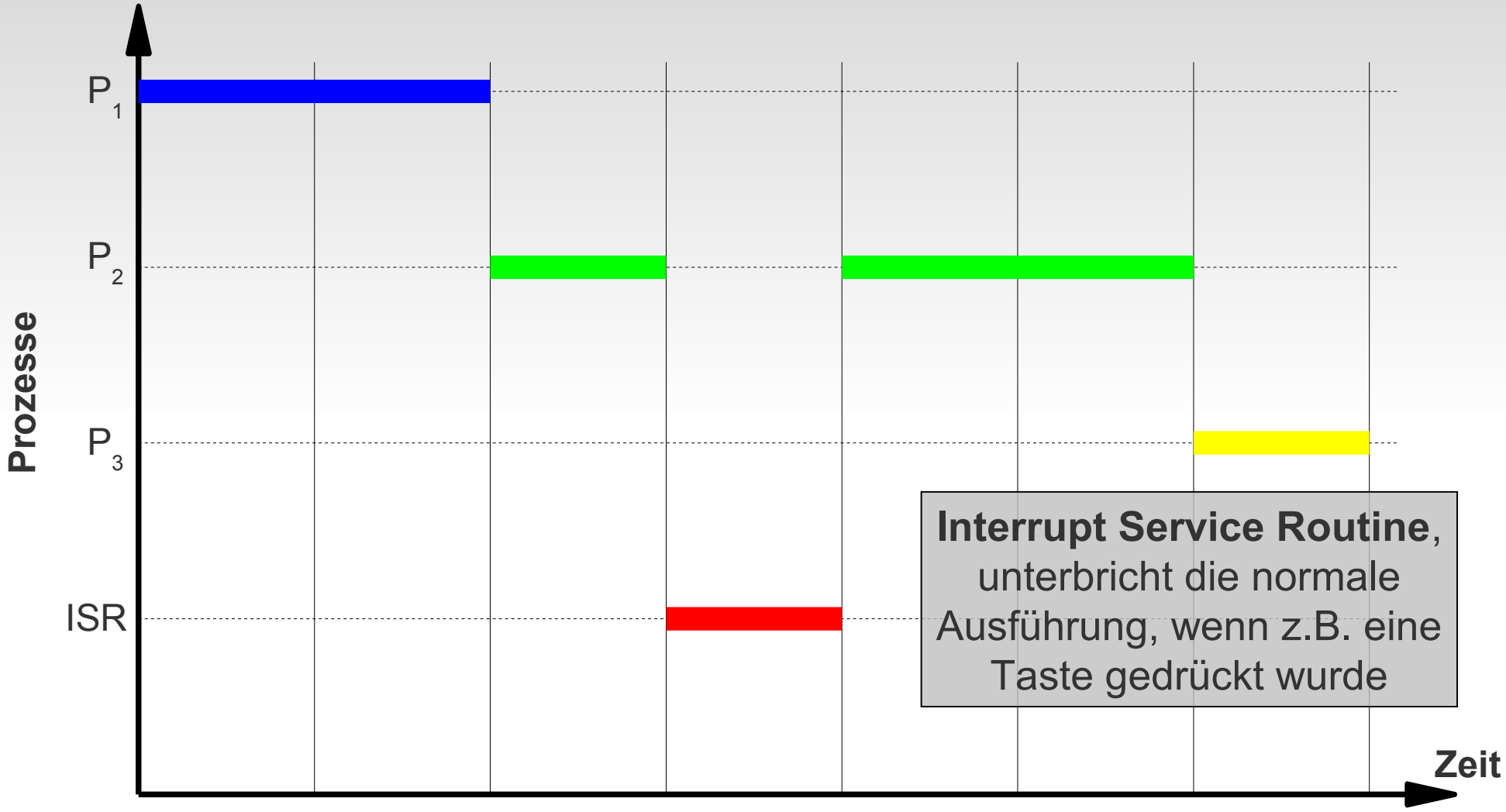
- Eigenschaften der Sprache Java
- Grundlagen objektorientierter Entwürfe
- Grafische Oberflächen mit AWT/Swing
- Nebenläufige Programmierung mit Threads
- Datei-Input/Output über Datenströme
- Client/Server-Programmierung mit Sockets
- Webtechnologien (HTTP, HTML, CSS, …)
- Servlets und JSPs im Java Webcontainer (Java EE)

Können Sie es wirklich noch?

Das wiederholen wir lieber noch einmal!

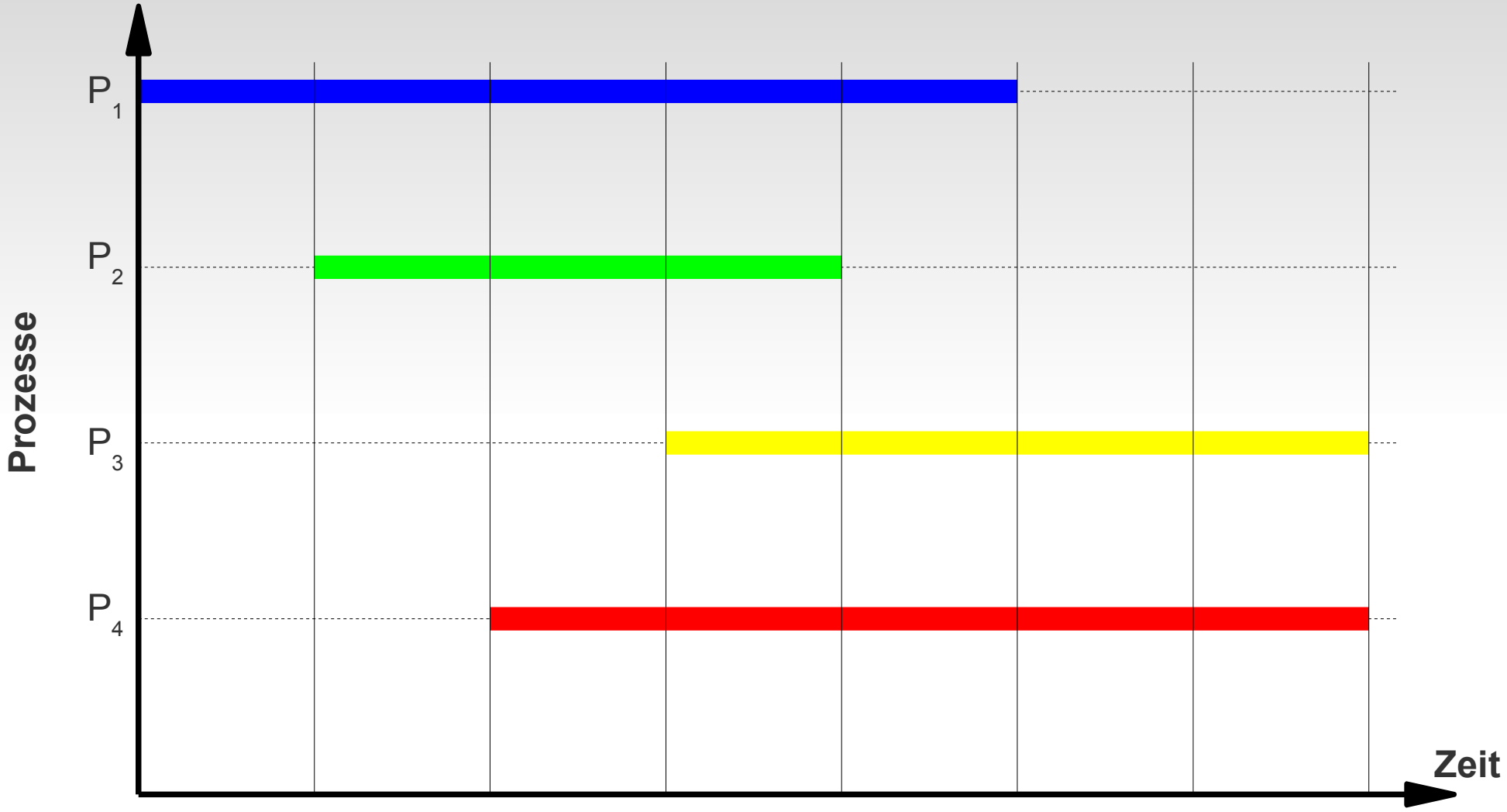
Nebenläufige Programmierung

Sequenzielle Ausführung



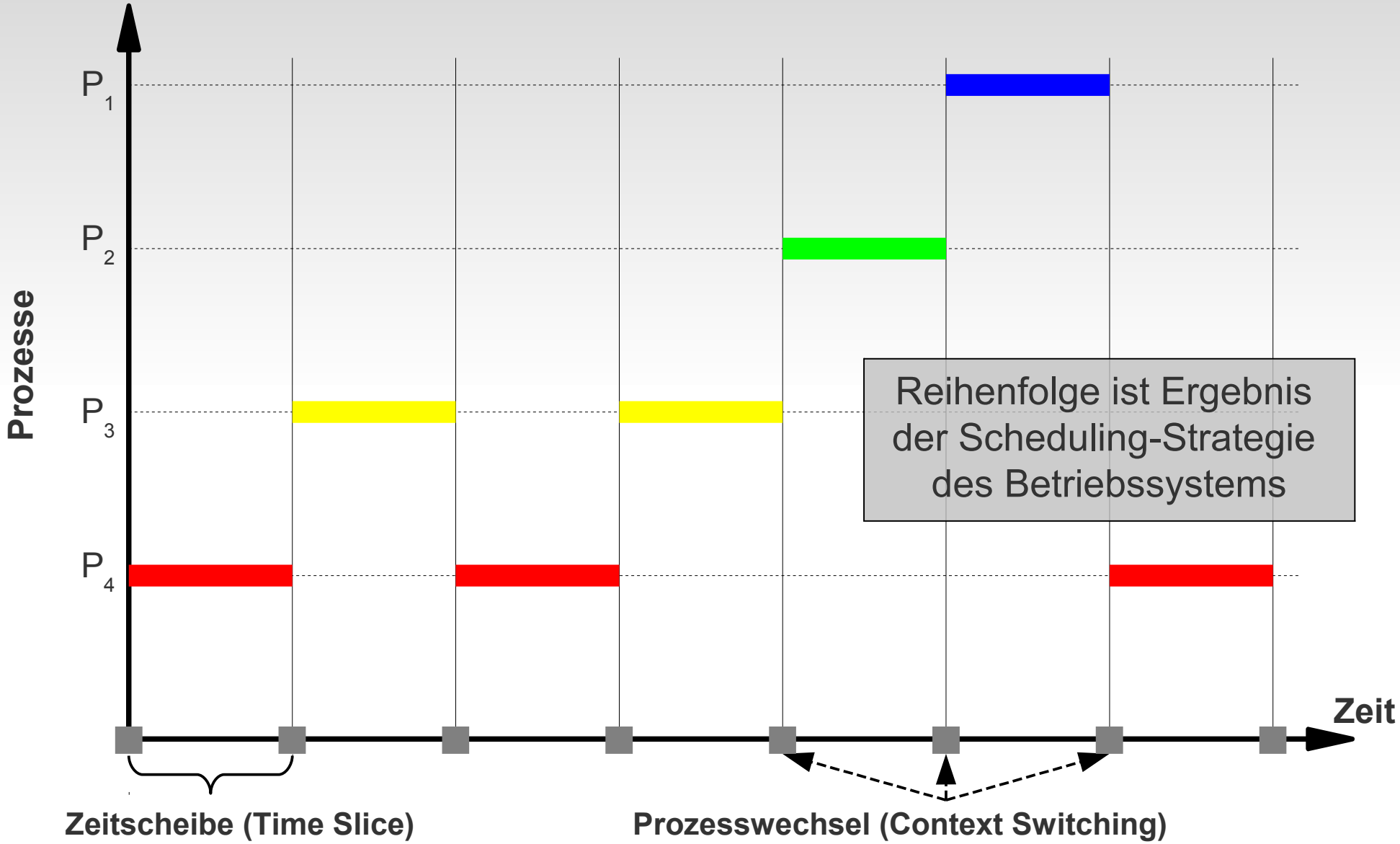
Frühe Betriebssysteme kannten keine Nebenläufigkeit
Je ein Prozess beanspruchte die volle Prozessorkapazität

Echt parallele Ausführung



Mehrere Prozesse laufen auf unterschiedlichen Prozessoren

Quasi parallele Ausführung



Nebenläufige Anwendungen

Nicht nur Anwendungen können parallel laufen

Anwendungen können selbst auch nebenläufig sein

- Mehr als ein Programmfluss zur selben Zeit
- Mehrere Vorgänge der Anwendung laufen parallel
- Oft notwendig, damit die Anwendung nicht einfriert
- Bessere Gliederung des Anwendungscodes möglich

Wird durch sogenannte Threads realisiert

- Teilen alle Betriebsmittel eines Prozesses
- Gliedern diesen in mehrere parallele Abläufe
- Werden darum auch leichtgewichtige Prozesse genannt

Nebenläufige Anwendungen

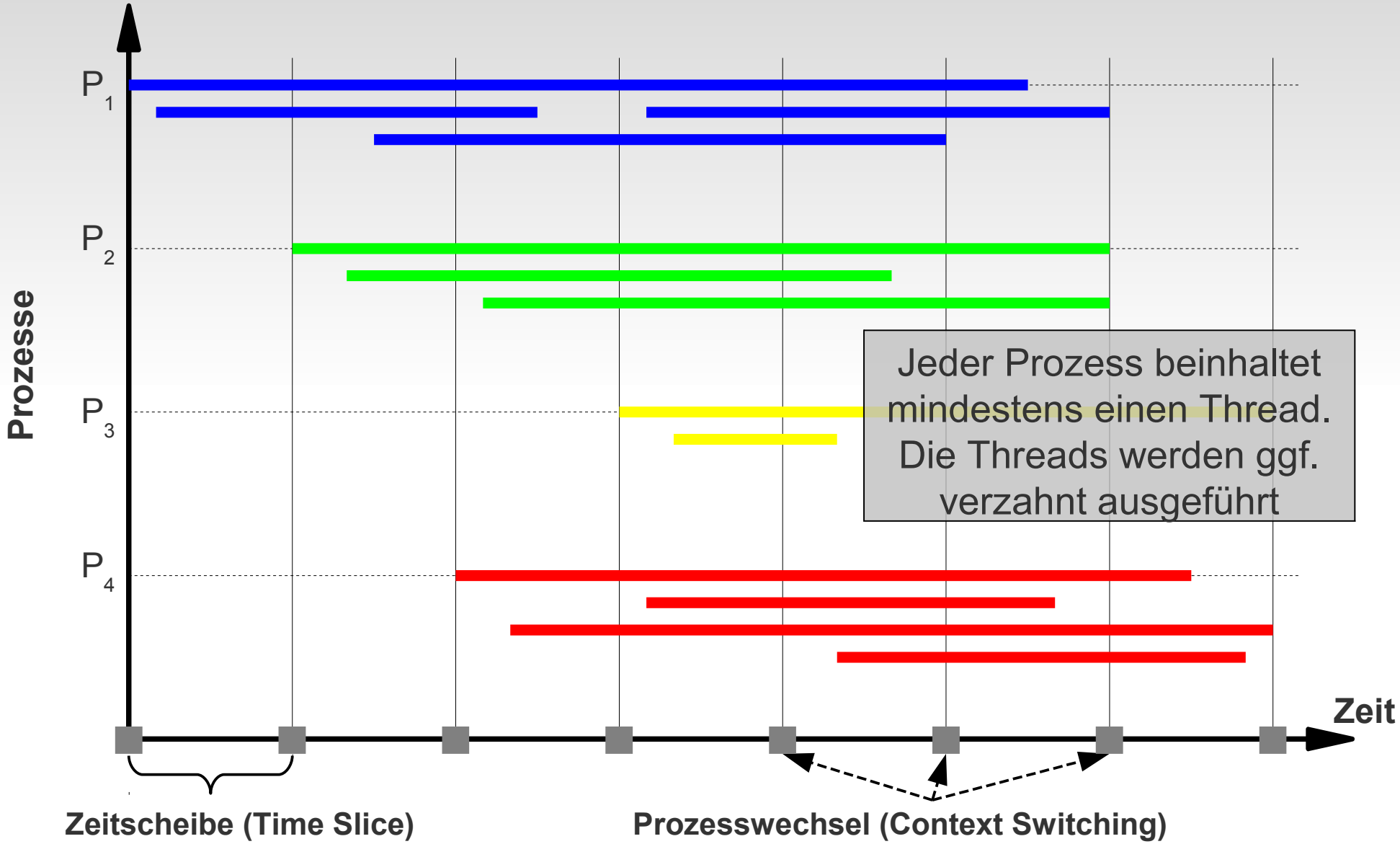
Typische Aufgaben eines Threads

- Steuerung der Benutzungsschnittstelle
- Langwierige Berechnungen
- Vorbereitung eines Druckauftrags
- Musikwiedergabe in Spielen oder Media Playern, …

In threadfähigen Betriebssystemen gilt

- Mindestens ein Thread je Prozess
- Threads kapseln die ausführbaren Rechnerbefehle

Mehrere Threads je Prozess



Threadunterstützung in Java

Fester Bestandteil der Sprache Java

Funktioniert auf jedem Betriebssystem

- Direkt in die Virtual Machine eingebaut
- Unabhängig von den Threadfunktionen des Systems
- **Bisher:** Keine Nutzung nativer Systemroutinen
- **Java 6:** Nutzung der Systemfunktionen wenn möglich, eigene Implementierung wenn nötig
- Kein Unterschied bei der Programmierung

Realisiert durch die Klasse **`java.lang.Thread`**

Beispiel: Threaderzeugung

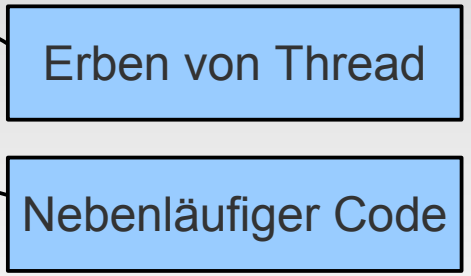
```
public class Main {
    public static void main(String[] args) {
        System.out.println("Knight Rider das Java-Programm");
        System.out.println("==============================");
        System.out.println();

        // Threadobjekte erzeugen
        Thread michael = new WorkerThread("Michael");
        Thread deavon = new WorkerThread("Deavon");
        Thread bonnie = new WorkerThread("Bonnie");
        Thread kit = new WorkerThread("Kit");

        // Threads starten
        michael.start();
        deavon.start();
        bonnie.start();
        kit.start();
    }
}
```

Beispiel: Threaderzeugung

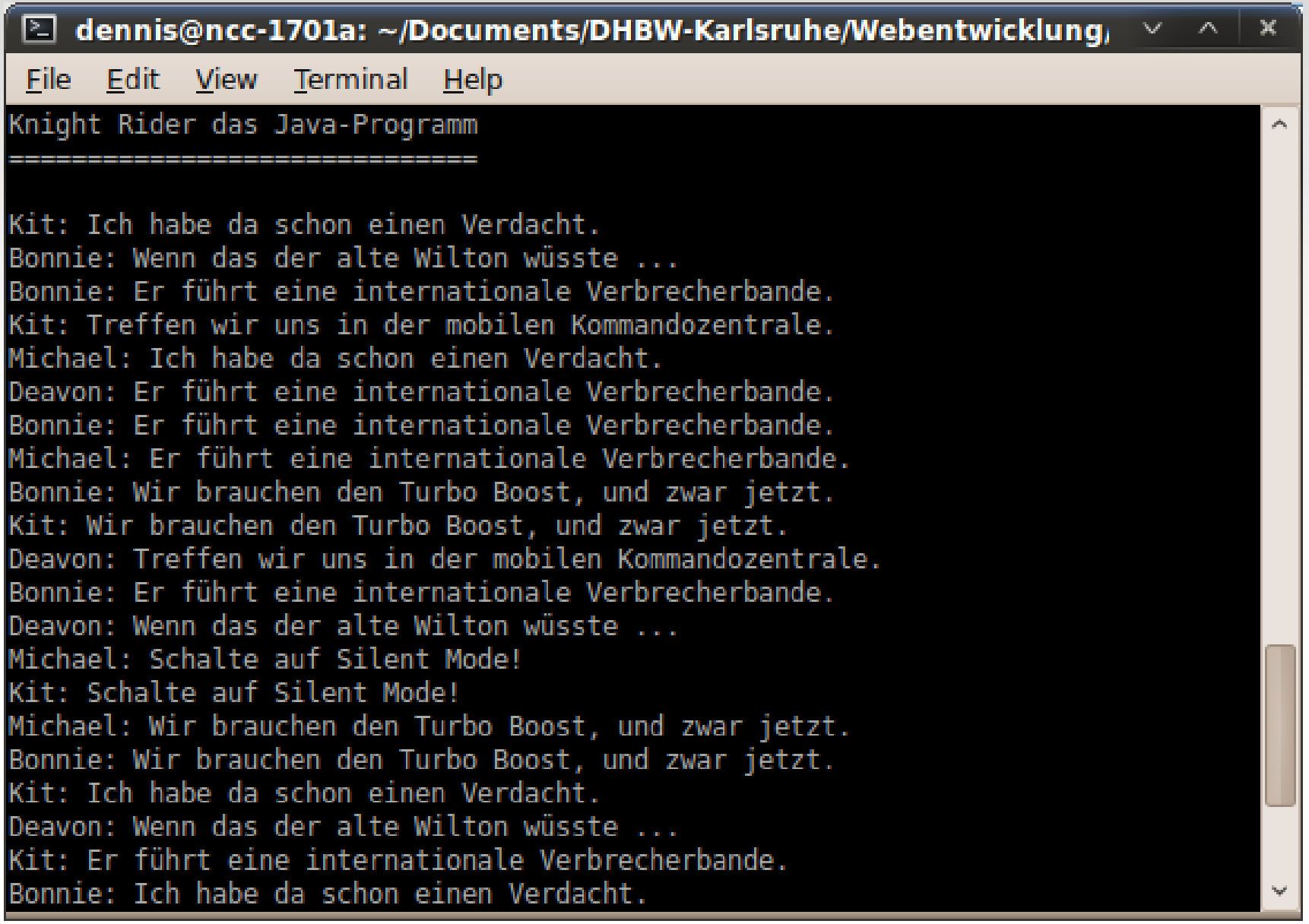
```
// Die Klasse muss von Thread erben, damit ihr
// Code nebenläufig ausgeführt werden kann

public class WorkerThread extends Thread {
    public WorkerThread(String name) {
        this.setName(name);
    }

    public void run() {
        // Hier kommt der nebenläufig
        // ausgeführte Code hin
    }

    // Nur die run()-Methode wird nebenläufig
    // ausgeführt. Aber auch nur dann, wenn sie
    // durch Aufruf von start() gestartet wurde!
}
```

Beispiel: Threaderzeugung

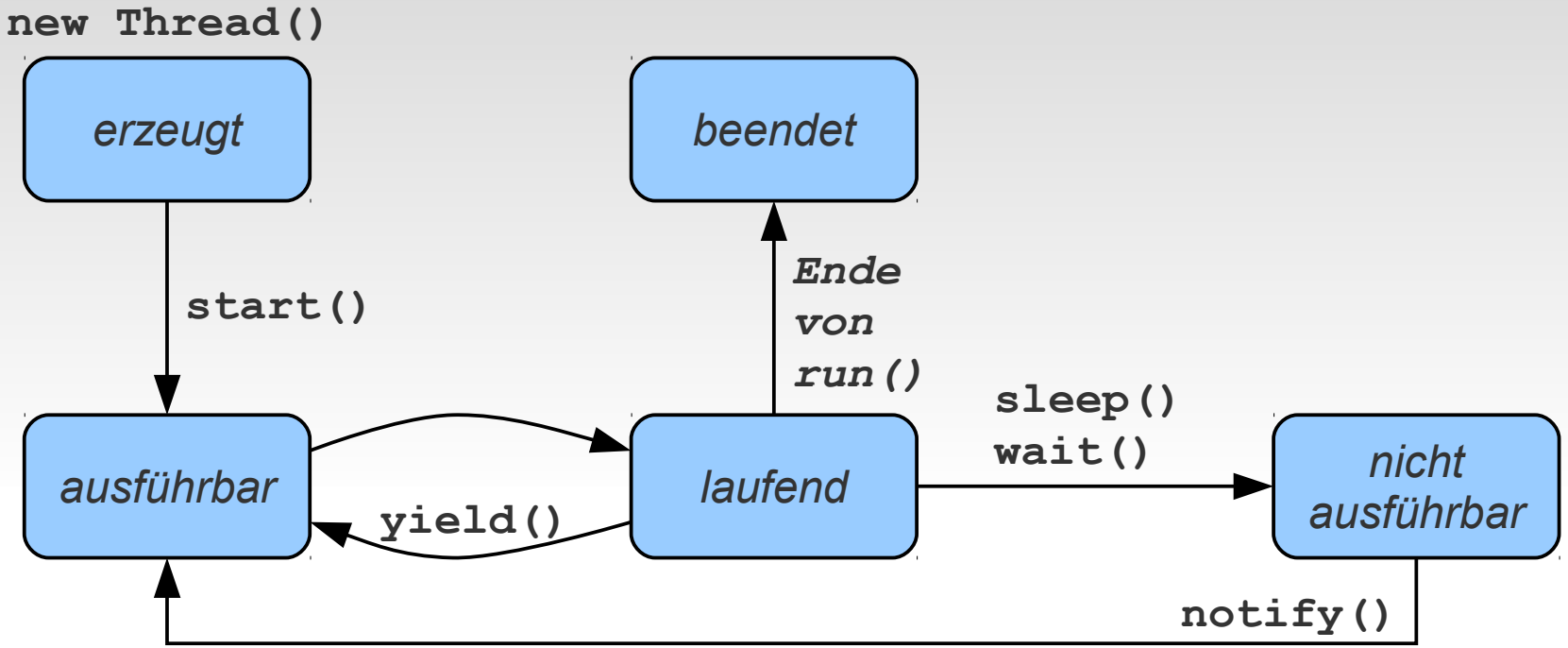
```
public class WorkerThread extends Thread {
    public WorkerThread(String name) {
        this.setName(name);
    }

    public void run() {
        while (!this.isInterrupted()) {
            try {
                Thread.sleep((int) (Math.random() * 2000) + 1000);
            } catch (InterruptedException ex) { }

            int nr = (int) (Math.random() * 6);
            String say = "";

            if (nr == 0) {
                say = "Wir brauchen den Turbo Boost, und zwar jetzt.";
            } // ...

            System.out.println(this.getName() + ": " + say);
        }
    }
}
```



Bildschirmausgabe



Zustände eines Threads

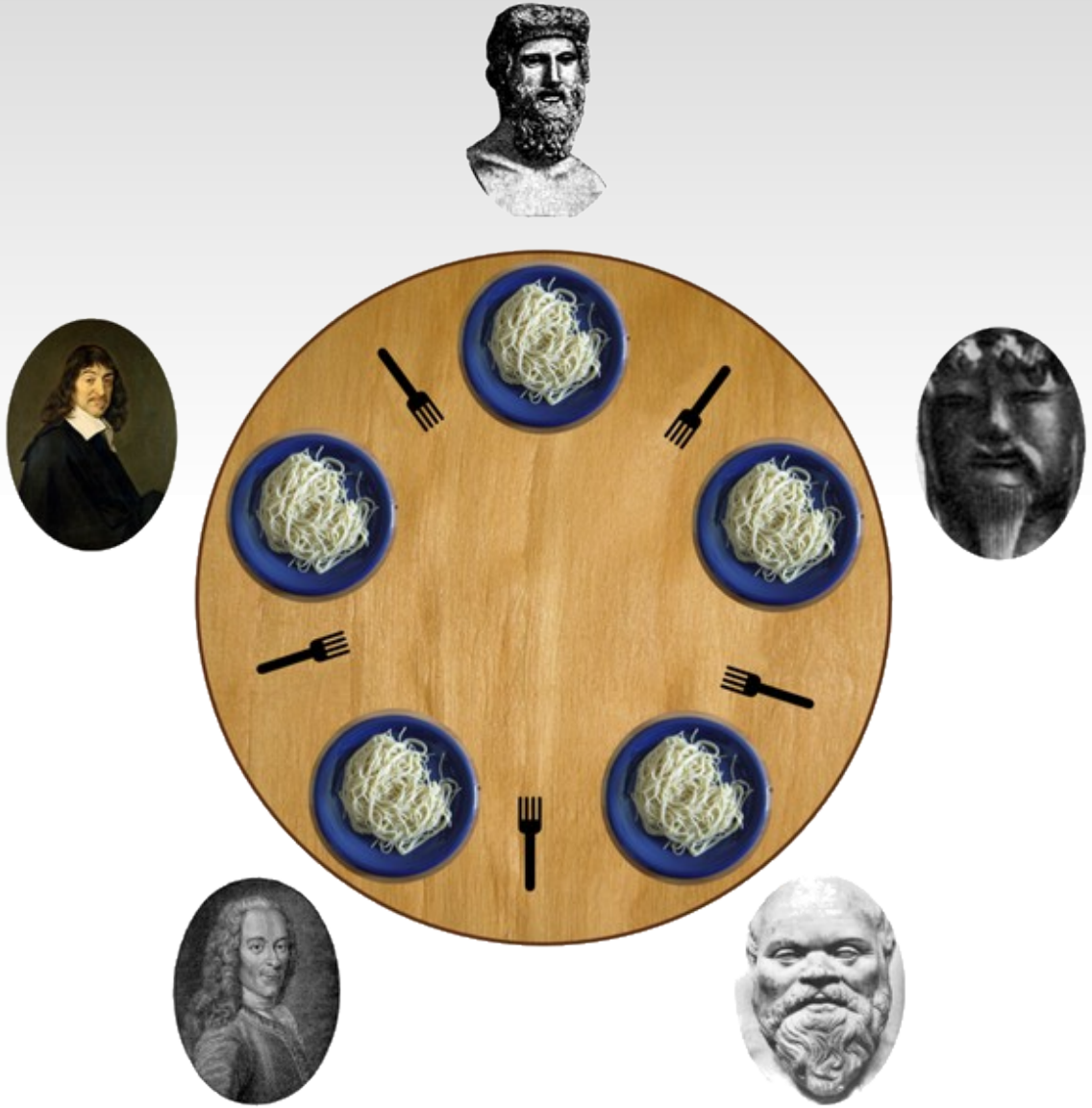


Ausführbar Thread kann eingeplant werden

Laufend Thread läuft gerade (nur einer)

Nicht ausführbar Thread wartet auf Betriebsmittel

Beispiel: Speisende Philosophen



Vorgehensweise

1. Denke eine Weile nach
2. Nimm die linke Gabel
3. Nimm die rechte Gabel
4. Esse, bis du satt bist
5. Lege beide Gabeln hin
6. Zurück zu Schritt 1

Geht das gut?

Verklemmung !?!

Die Philosophen müssen verhungern

Wettlaufsituationen

Lost Update-Problem

- Zwei Threads lesen und schreiben dasselbe Feld
- Ein Thread überschreibt die Änderungen des Anderen

Leser/Schreiber-Problem / Dirty Read-Problem

- Ein Thread schreibt Daten in ein Feld
- Ein anderer Thread liest Daten aus demselben Feld

Erzeuger/Verbraucher-Problem

- Ein Thread konsumiert was ein anderer produziert
- Konsument muss auf Produzent warten

Nebenläufige Threads, Verteilte Datenbankanwendungen, …

Thread-Synchronisation

Kritische Abschnitte: Dürfen nicht parallel laufen

- Keine gleichzeitige Ausführung kritischer Routinen
- Ausführung nur unter **gegenseitigem Ausschluss**
- Hierfür Verwendung sogenannter Mutex-Primitive

Synchronisation konkurrierender Threads

- Meist beliebige Sequenzialisierung möglich
- Solange keine kritischen Abschnitte gleichzeitig laufen

Synchronisation kooperierender Threads

- Einzuhaltende Reihenfolge der Threadausführung
- Mit Mutex-Primitiven allein nicht realisierbar

Beispiel: Konkurrierende Threads

Thread 1: myCounter: 4

```
int myCounter = counter;
myCounter++;
counter = myCounter;
```

Thread 2: myCounter: 4

```
int myCounter = counter;
myCounter++;
counter = myCounter;
```

Gemeinsame Variable

counter: 4

Mögliche Ausführung

```
T1: int myCounter = counter;
T1: myCounter++;
T1: counter = myCounter;
T2: int myCounter = counter;
T2: myCounter++;
T2: counter = myCounter;
```

✓

```
T1: int myCounter = counter;
T2: int myCounter = counter;
T2: myCounter++;
T1: myCounter++;
T1: counter = myCounter;
T2: counter = myCounter;
```

✗

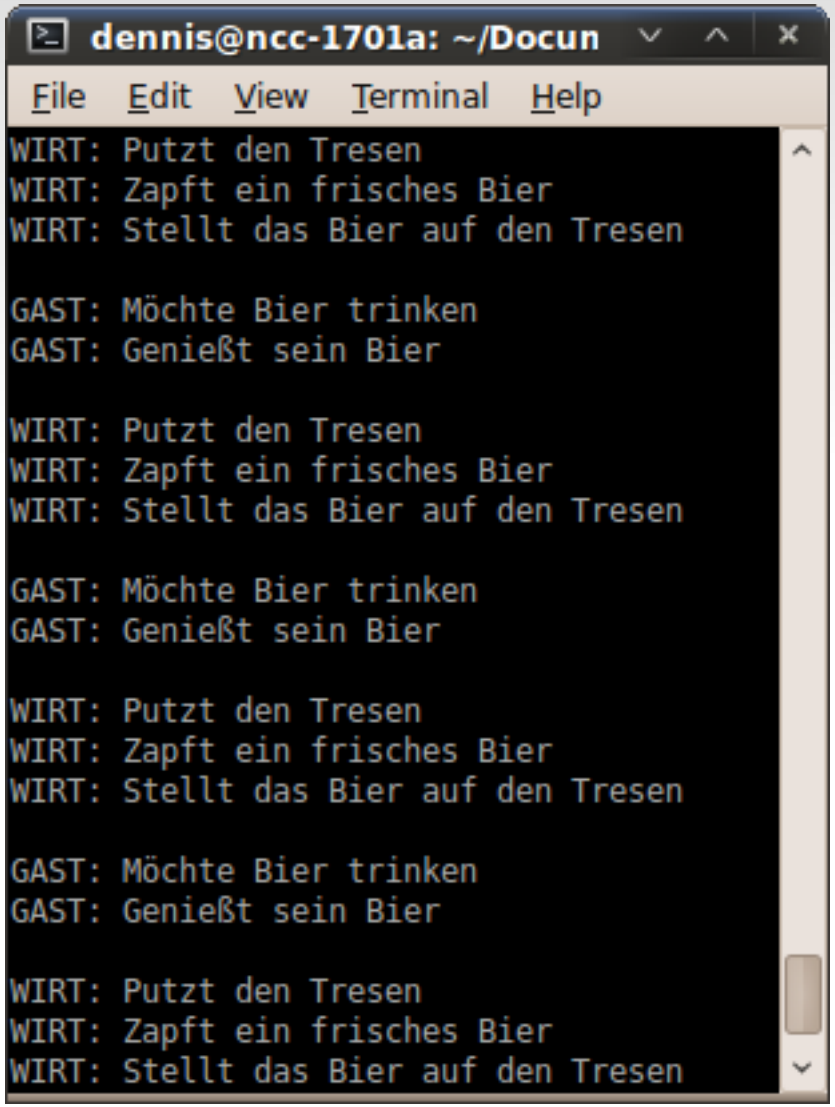
Beipsiel: Kooperierende Threads

Barkeeper produziert (zapft) ein Bier ...

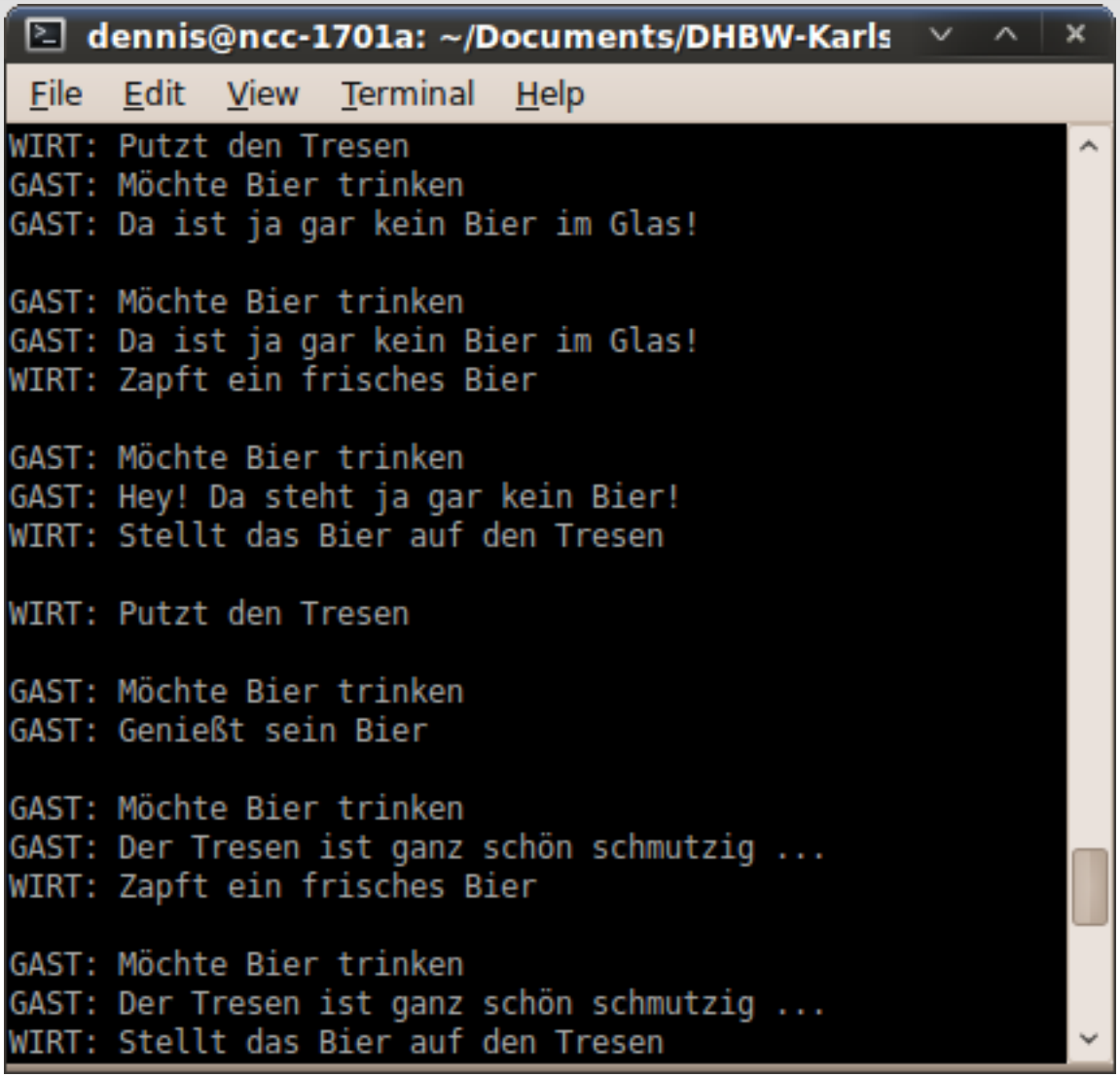
Kunde konsumiert es

Problem: Produzent/Konsument

Gewünschtes Verhalten



Tatsächliche Ausgabe des Programms



Streams und Sockets

Was ist ein Datenstrom?

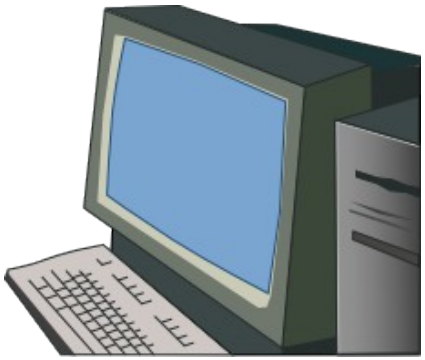
Ein Datenstrom besteht aus …

- Einer Menge gleichartiger Daten,
- Auf die nur sequentiell zugegriffen werden kann,
- Die also nur fortlaufend verarbeitet werden kann
- Und deren Ende im Voraus nicht absehbar ist

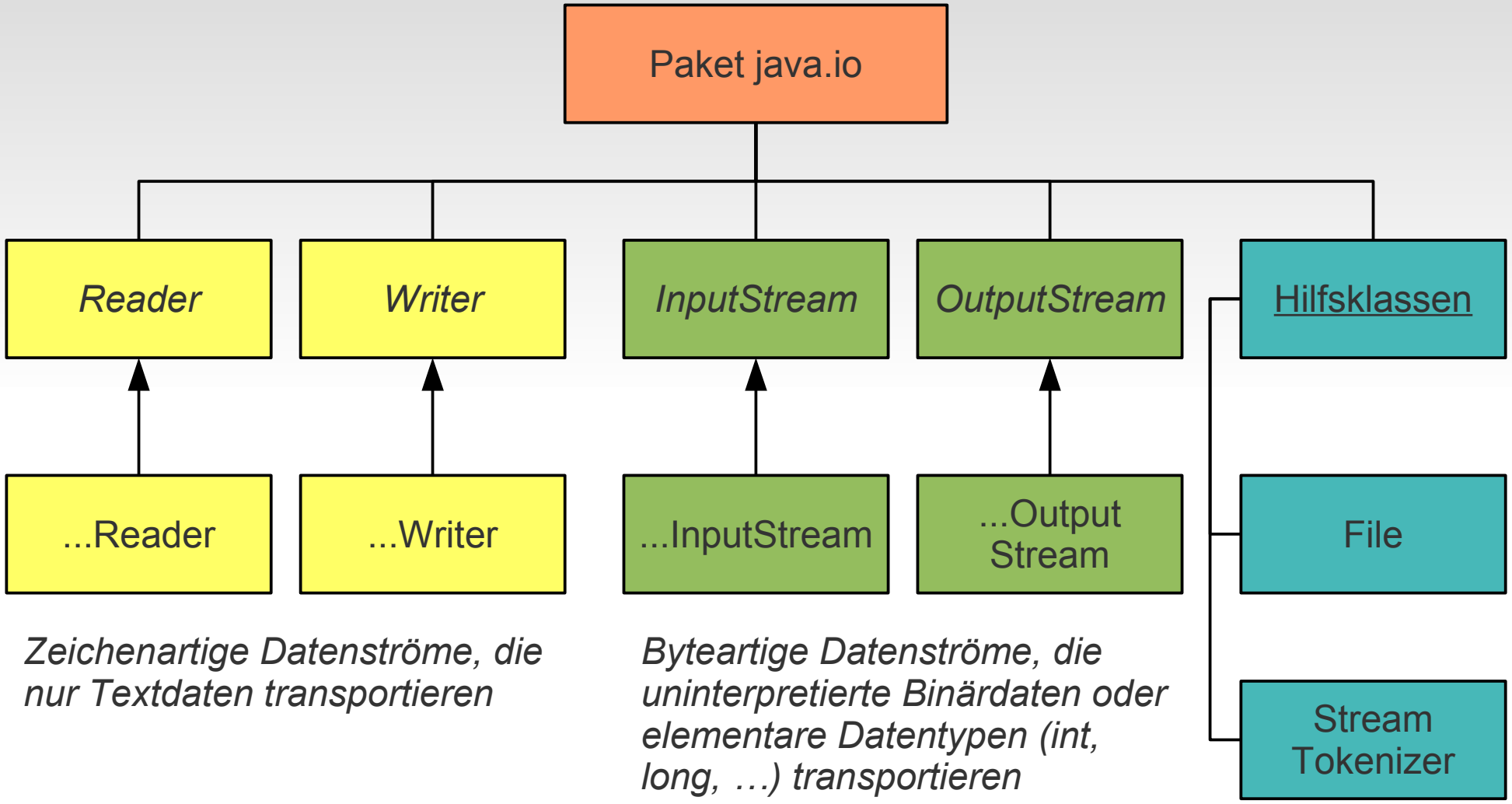
Besitzt eine Quelle und ein Ziel

Gegenteil: Strukturen mit wahlfreiem Zugriff

>>> 1011010010 >>>



Das Paket java.io



Zeichenartige Datenströme, die nur Textdaten transportieren

Byteartige Datenströme, die uninterpretierte Binärdaten oder elementare Datentypen (int, long, …) transportieren

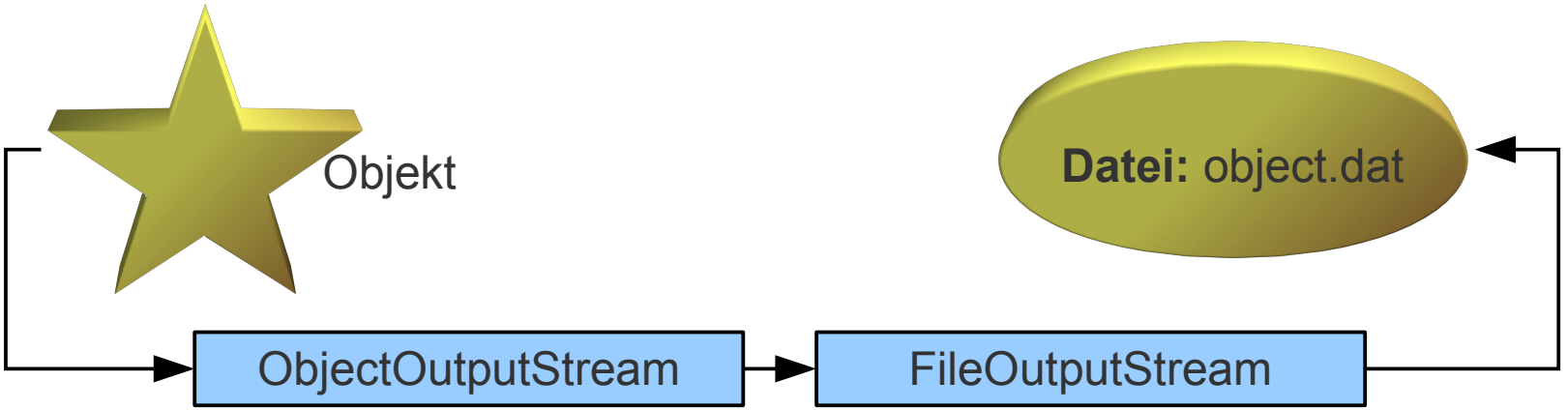
Datenströme in Java

Byteströme (`InputStream`, `OutputStream`)

- Umfassen entweder uninterpretierte Bytes,
- Elementare Typen (int, float, double, …)
- Oder serialisierte Objektzustände

Zeichenströme (`Reader`, `Writer`)

- Umhüllen einen **`InputStream`** oder **`OutputStream`**
- Interpretieren den Strom gemäß einem Zeichensatz



Methoden der Klasse Reader

`public abstract int read()`

- Liefert das nächste Zeichen des Stroms in Form eines **`int`**-Werts
- Gibt **`-1`** zurück, wenn keine Daten mehr vorhanden sind

`public int read(char[] c)`

Befüllt das Character Array mit den nächsten Zeichen des Stroms

`public int read(char[] c, int off, int n)`

Wie eben, nur mit Positions- und Längenangabe innerhalb des Stroms

`public void close()`

Schließt den Strom. Danach kann nichts mehr gelesen werden

Methoden der Klasse Writer

```
public abstract void write(int c)
```

Schreibt das in `c` übergebene Zeichen in den Datenstrom

```
public void write(char[] c)
```

Schreibt die Zeichen des Character Arrays in den Datenstrom

```
public void write(char[] c, int off, int n)
```

Wie eben, nur mit Positions- und Längenangabe innerhalb des Stroms

```
public void write(String s)
public void write(String s, int off, int n)
```

Schreibt die im String enthaltenen Zeichen in den Strom

Methoden der Klasse Writer

`public abstract void close()`

Löst ein **`flush()`** aus und schließt den Strom

`public void flush()`

- Arbeitet alle Zeichen ab, die noch im Puffer des Stroms sind
- Schiebt den Inhalt des internen Puffers in den Datenstrom

Reader/Writer-Klassen

Umwandlung von Byteströmen zu Zeichenströmen

- `InputStreamReader`
- `OutputStreamWriter`

Lesen und Schreiben von Textdateien

- `FileReader`
- `FileWriter`

Gepuffertes Lesen und Schreiben anderer Ströme

- `BufferedReader`
- `BufferedWriter`

Formatierung von ausgegebenen Textzeilen

- `PrintWriter`

Beispiel: Textdateien kopieren

```
import java.io.*;

public class CopyTextFile {
    public static void main(String[] args) {
        File srcFile = null;
        File dstFile = null;

        // File-Objekte erzeugen
        try {
            srcFile = new File(args[0]);
            dstFile = new File(args[1]);
        } catch (ArrayIndexOutOfBoundsException ex) { ... }

        // Datei kopieren
        try {
            CopyTextFile.execute(srcFile, dstFile);
        } catch (IOException ex) { ... }
    }
    ...
```

Beispiel: Textdateien kopieren

```
// Datei kopieren und auf Konsole ausgeben
// Dabei auf IOExceptions aufpassen
public static void execute(File srcFile, File dstFile)
throws IOException {
    Reader fromSrcFile = new FileReader(srcFile);
    Writer toDstFile = new FileWriter(dstFile);
    Writer toScreen = new OutputStreamWriter(System.out);
    int c;

    while ((c = fromSrcFile.read()) != -1) {
        toDstFile.write(c);
        toScreen.write(c);
    }

    fromSrcFile.close();
    toDstFile.close();
    toScreen.close();
    }
}
```

Hier kann überall eine **IOException** auftreten.

Methoden der Klasse InputStream

`public abstract int read()`

- Liefert das nächste Byte aus dem Strom als Integer-Wert
- Liefert `-1`, wenn das Stromende erreicht wurde

`public int read(byte[] b)`

- Befüllt das übergebene Array mit den nächsten Bytes des Stroms
- Liefert `-1`, wenn das Ende des Datenstroms erreicht wurde

`public int read(byte[] b, int off, int n)`

Wie eben, nur mit Positions- und Längenangabe innerhalb des Stroms

`public void close()`

Schließt den Strom, so dass keine Daten mehr gelesen werden können

Methoden der Klasse InputStream

`public abstract int read()`

- Liefert das nächste Byte aus dem Strom als Integer-Wert
- Liefert `-1`, wenn das Stromende erreicht wurde

`public int read(byte[] b)`

- Befüllt das übergebene Array mit den nächsten Bytes des Stroms
- Liefert `-1`, wenn das Ende des Datenstroms erreicht wurde

`public int read(byte[] b, int off, int n)`

Wie eben, nur mit Positions- und Längenangabe innerhalb des Stroms

`public void close()`

Schließt den Strom, so dass keine Daten mehr gelesen werden können

Methoden der Klasse OutputStream

```
public abstract void write(int b)
```

Schreibt das in `c` übergebene Byte in den Datenstrom

```
public void write(byte[] b)
```

Schreibt die Bytes des übergebenen Byte Arrays in den Datenstrom

```
public void write(byte[] b, int off, int n)
```

Wie eben, nur mit Positions- und Längenangabe innerhalb des Stroms

```
public void close()
```

Schließt den Strom. Anschließend kein Schreiben mehr möglich.

```
public void flush()
```

Arbeitet den eventuell vorhandenen Puffer ab und leer ihn.

InputStream/OutputStream-Klassen

Lesen und Schreiben von Binärdateien

- `FileInputStream`
- `FileOutputStream`

Gepuffertes Lesen und Schreiben anderer Ströme

- `BufferedInputStream`
- `BufferedOutputStream`

Lesen und Schreiben elementarer Datentypen

- `DataInputStream`
- `DataOutputStream`

Lesen und Schreiben serialisierter Objekte

- `ObjectInputStream`
- `ObjectOutputStream`

Ganze Objekte versenden

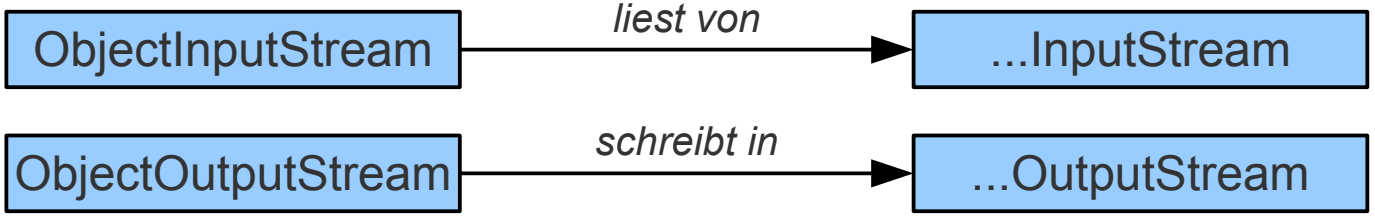
Zwei Klassen, um Objekte zu lesen und zu schreiben

- **`ObjectInputStream`**
- **`ObjectOutputStream`**

Können auch elementare Werte lesen und schreiben vgl. mit **`DataInputStream`**, **`DataOutputStream`**

Zusätzlich können Objekte (de)serialisiert werden

Hierfür müssen die Objekte das Markerinterface **`Serializable`** implementieren



Methoden zur (De)serialisierung

ObjectInputStream

```
public final Object readObject()
```

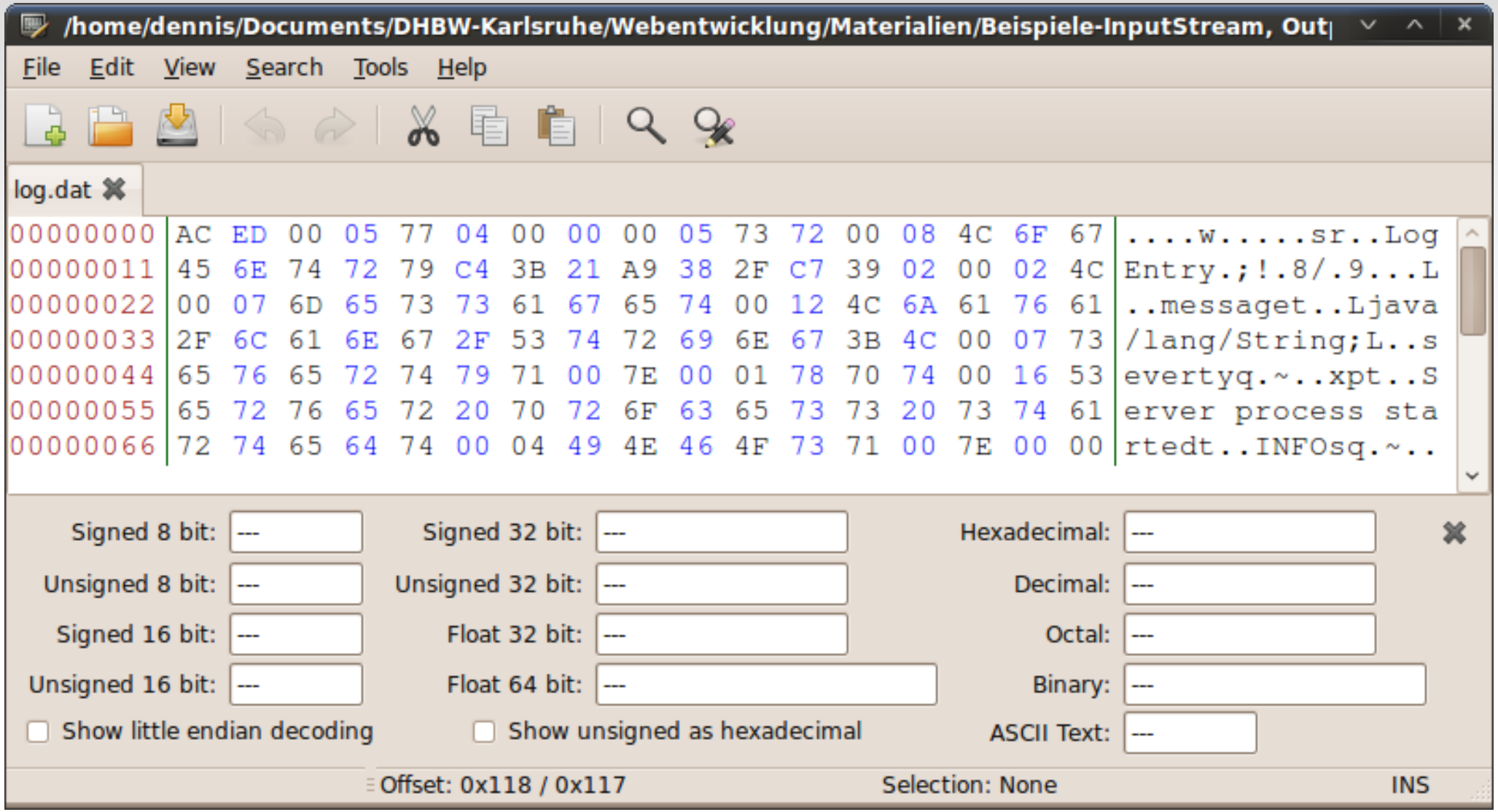
- Liest ein Objekt aus dem Datenstrom und deserialisiert es
- Für das zurückgegebene Objekt wird ein Typecast benötigt
- Die Klasse des serialisierten Objekts muss vorhanden sein, damit das Objekt deserialisiert werden kann!
- Kann daher eine `ClassNotFoundException` auslösen

ObjectOutputStream

```
public final void writeObject(Object obj)
```

Serialisiert das übergebene Objekt und schreibt es in den Strom

Beispiel: Objektserialisierung



Netzwerkprotokolle

An der Kommunikation beteiligte Protokolle

- Das **Anwendungsprotokoll** legt die ausgetauschten Daten und das Verhalten der Anwendungen fest
- Die **Netzwerkprotokolle** dienen der technischen Abwicklung und sind nicht Teil der Anwendungen

Das OSI-Modell als einzig relevanter Standard

- Umfasst sieben Protokollschichten
- **Oberste Schicht:** Das Anwendungsprotokoll
- **Unterste Schicht:** Physikalisch/Elektrische Schicht

Das vereinfachte OSI-Modell

Vier Schichten für die Programmierung

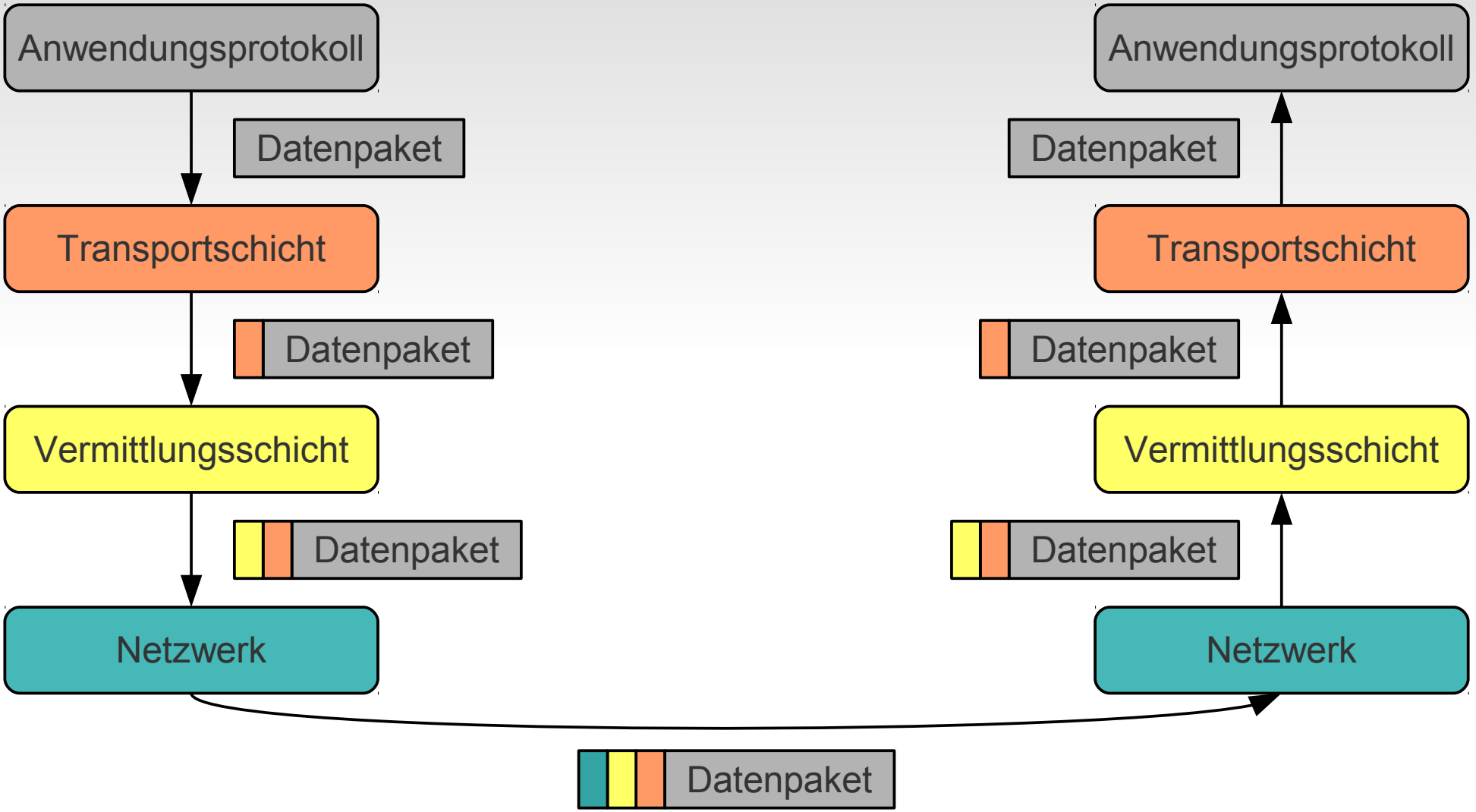
- Anwendungsprotokoll
- Transportschicht (TCP oder UDP)
- Vermittlungsschicht (IP)
- Physikalisches Netzwerk

Die Daten durchlaufen stets sämtliche Schichten

Jede Schicht fügt dem Datenpaket Informationen zu

Alle Schichten sind unabhängig und austauschbar

Wechselspiel der Schichten



Aufgaben der Schichten

Physikalische Netzwerkschicht

- Beinhaltet die elektrische Verbindung der Systeme
- Beispiel: CAT5-Kabel, Glasfaser oder Richtfunk
- Beschreibt die Codierung der digitalen Zustände

Vermittlungsschicht

- Wird durch das Internet Protokoll (IP) dargestellt
- Ermöglicht das Senden/Empfangen von Datenpaketen
- Legt fest, wie Pakete ihr Ziel finden (Paket Switching)
- Keine Zusagen hinsichtlich der Verbindungsgüte

Aufgaben der Schichten

Transportschicht

- Multi-Plexing der Rechneradresse durch Ports
- Ermöglicht den Aufbau von logischen Verbindungen
- Sichert u.U. Fehlerkorrektur und die Zustellung der Pakete in der richtigen Reihenfolge zu (TCP)
- UDP hingegen macht keine solche Zusagen

Anwendungsprotokoll

- Beschreibt die ausgetauschten Daten
- Ist nicht die Anwendung selbst !!

Socket-Programmierung

Sockets stellen die Endpunkte der Netzwerkkommunikation dar

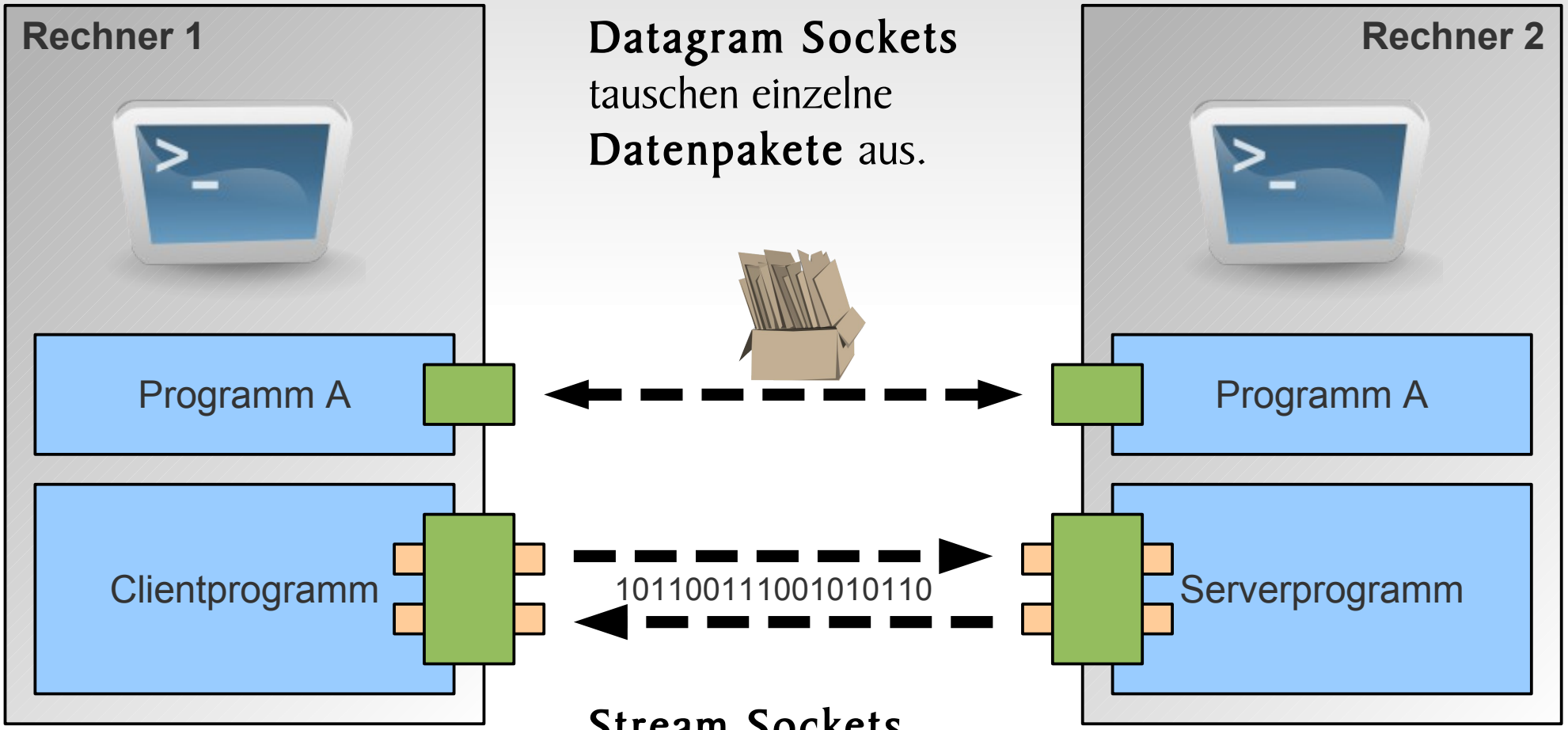
Grundsätzlich zwei Arten Sockets

- **Stream Sockets:** Dienen dem bidirektionalen Datenaustausch
- **Datagram Sockets:** Dienen dem Versand einzelner Datenpakete

Sie bilden die Schnittstelle zu den Netzwerkfunktionen des Systems

Ursprünglicher Entwurf ähnlich der Unix File APIs (BSD, 1983)

Beispiel: Socket-Arten



Datagram Sockets tauschen einzelne **Datenpakete** aus.

Stream Sockets nutzen kontinuierliche Datenströme.

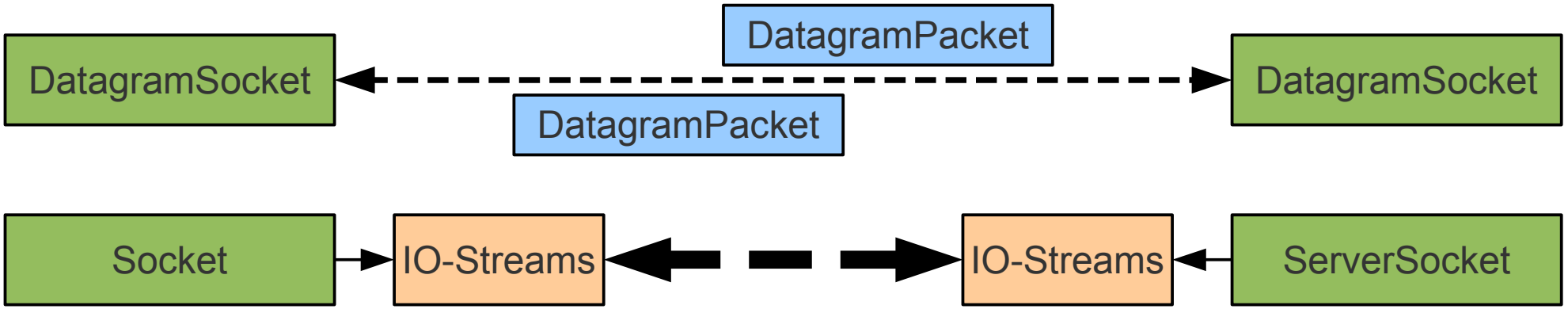
Socket-Klassen

Austausch einzelner UDP-Datagramme

- **Client/Server:** `DatagramSocket`
- **Datenpakete:** `DatagramPacket`

Sockets für Bidirektionale TCP-Datenströme

- **Clientseite:** `Socket`
- **Serverseite:** `ServerSocket`



Methoden der Klasse Socket

```
public Socket(InetAddress adr, int port)
public Socket(String host, int port)
```

Erzeugung eines neuen Client Stream Sockets

```
public InputStream getInputStream()
```

Liefert den Byte-Eingabestrom des Sockets, um Daten zu lesen

```
public OutputStream getOutputStream()
```

Liefert den Byte-Ausgabestrom des Sockets, um Daten zu versenden

```
public void close()
```

Trennt die Verbindung und schließt alle Datenströme

Methoden von ServerSocket

`public ServerSocket(int port)`

Erzeugt einen neuen Server Socket für den gegebenen TCP-Port

`public ServerSocket(int port, int backlog)`

Erzeugen eines Server Sockets für maximal **`backlog`** Verbindungen

`public Socket accept()`

- Wartet darauf, dass ein Client sich mit dem Socket verbindet
- Erzeugt ein **`Socket`**-Objekt, das die neue Verbindung darstellt

`public void close()`

Trennt die Verbindung mit allen Clients und schließt alle Ströme

Zeit für ein Beispiel

Client-Programm

1. Öffne einen Socket in Verbindung mit dem Server.
2. Öffne einen Eingabestrom und einen Ausgabestrom über den Socket.
3. Lies von und schreibe auf die Ströme gemäß dem *Serverprotokoll*.
4. Schließe die Ströme.
5. Schließe den Socket.

Server-Programm

1. Öffne einen Server-Socket
2. Warte bis sich ein Client anmeldet.
3. Öffne einen Socket in Verbindung mit dem Client.
4. Öffne einen Eingabestrom und einen Ausgabestrom über den Client-Socket.
5. Lies von und schreibe auf die Ströme gemäß dem *Serverprotokoll*.
6. Schließe die Ströme.
7. Schließe den Socket.
8. Schließe den Server-Socket (oder gehe zu Schritt 2).

Beispiel: Client-Anwendung

```
import java.io.*;
import java.net.*;

public class Client {
    public static void main(String[] args) {
        try {
            Socket socket = new Socket("dhbw.de", 8080);
            InputStream fromServer = socket.getInputStream();
            OutputStream toServer = socket.getOutputStream();

            // Kommunikation gemäß Anwendungsprotokoll

            socket.close();

        } catch (UnknownHostException ex) {
            // Hostname kann nicht aufgelöst werden
        } catch (IOException ex) {
            // Allgemeiner Kommunikationsfehler
        }
    }
}
```

Kommunikation erfolgt einfach durch Nutzung der Streams.

Beispiel: Einfacher Server

```
import java.io.*;
import java.net.*;

public class Server {
    public static void main(String[] args) {
        try {
            ServerSocket serverSocket = new ServerSocket(8080);
            Socket socket = serverSocket.accept();

            InputStream fromClient = socket.getInputStream();
            OutputStream toClient = socket.getOutputStream();

            // Kommunikation gemäß Anwendungsprotokoll

            serverSocket.close();
        } catch (IOException ex) {
            // Kommunikationsfehler
        }
    }
}
```

Kommunikation erfolgt einfach durch Nutzung der Streams.

Dieser Server ist eigentlich recht nutzlos, denn er kann immer nur einen Client bedienen …

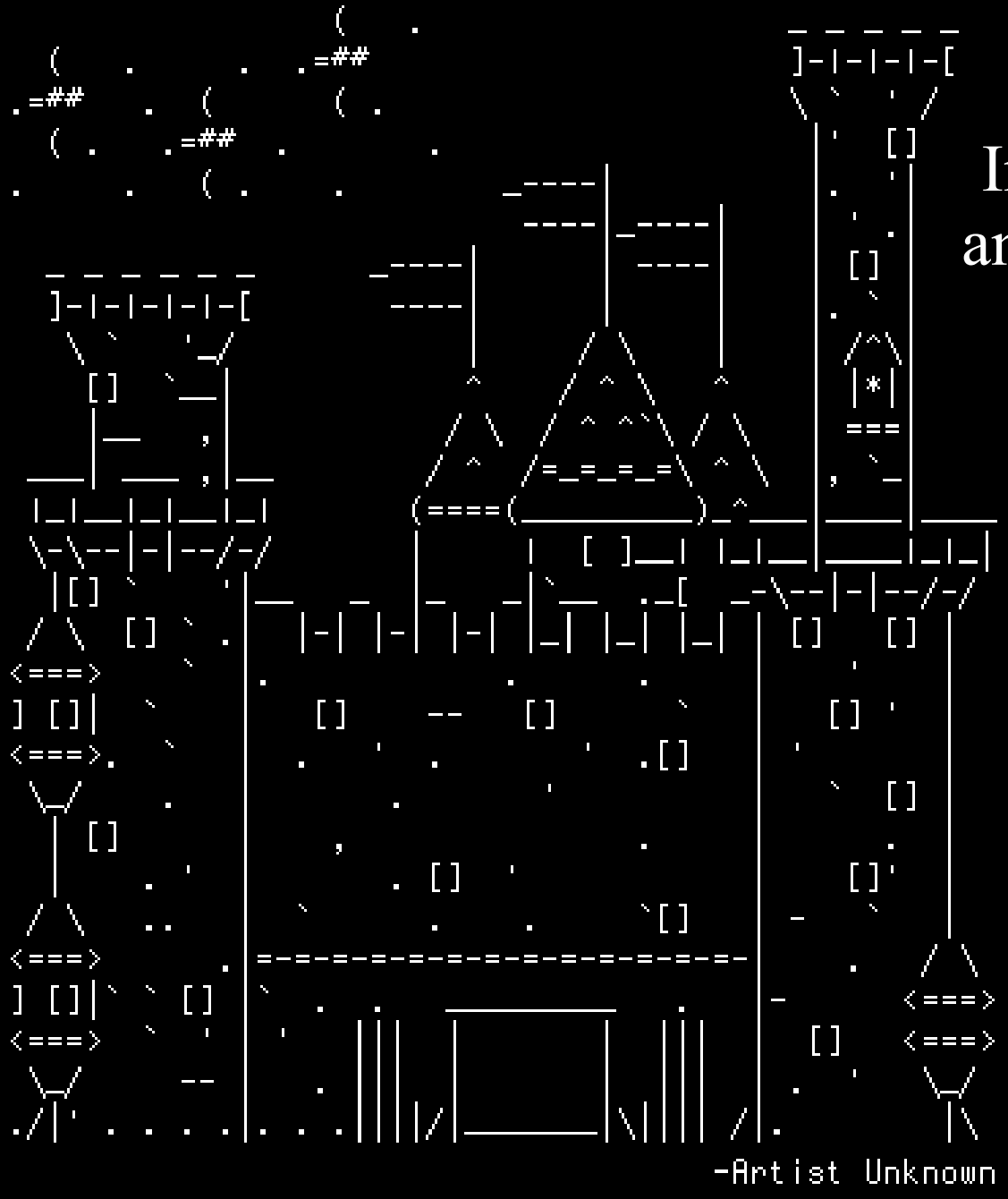
Beispiel: Server mit Threads

```
try {
    ServerSocket serverSocket = new ServerSocket(8080);
} catch (IOException ex) {
    System.exit(1);
}
```

```
while (true) {
    try {
        Socket socket = serverSocket.accept();
        InputStream fromClient = socket.getInputStream();
        OutputStream toClient = socket.getOutputStream();

        Thread handler = new HandlerThread(
            fromClient,
            toClient
        );
        handler.start();
    } catch (IOException ex) { ... }
}
```

Die Kommunikation mit dem Client wird an einen Thread delegiert. Das Hauptprogramm wartet dann schon auf die nächste Verbindung.


PLACE
STAMP
HERE
GOOGLE
CLASSIC
QUERY:
IMAGES
NEWS
VIDEO
MAPS
OTHER
SEND YOUR QUERY TO: GOOGLE INC., 1600 AMPHITHEATRE PARKWAY, MOUNTAIN VIEW, CA 94043, UNITED STATES
PLEASE ALLOW 30 DAYS FOR SEARCH RESULTS

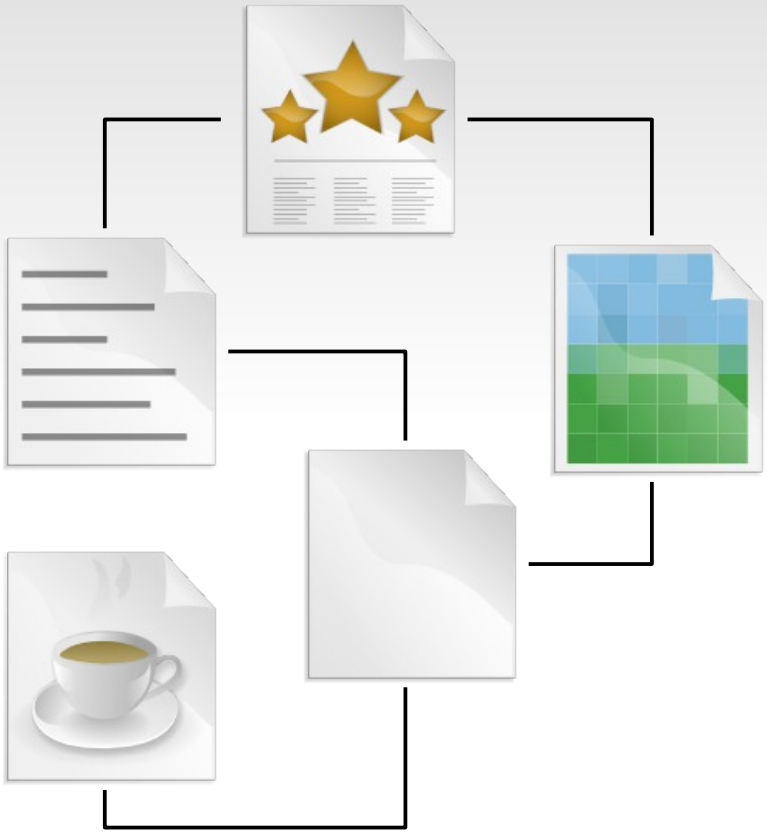


In the beginning was the Word,
and the Word was *ascii* encoded,
typically displayed in

monochrome courier
on a black screen.

-- dejavu.org

Eine einfache Idee ...

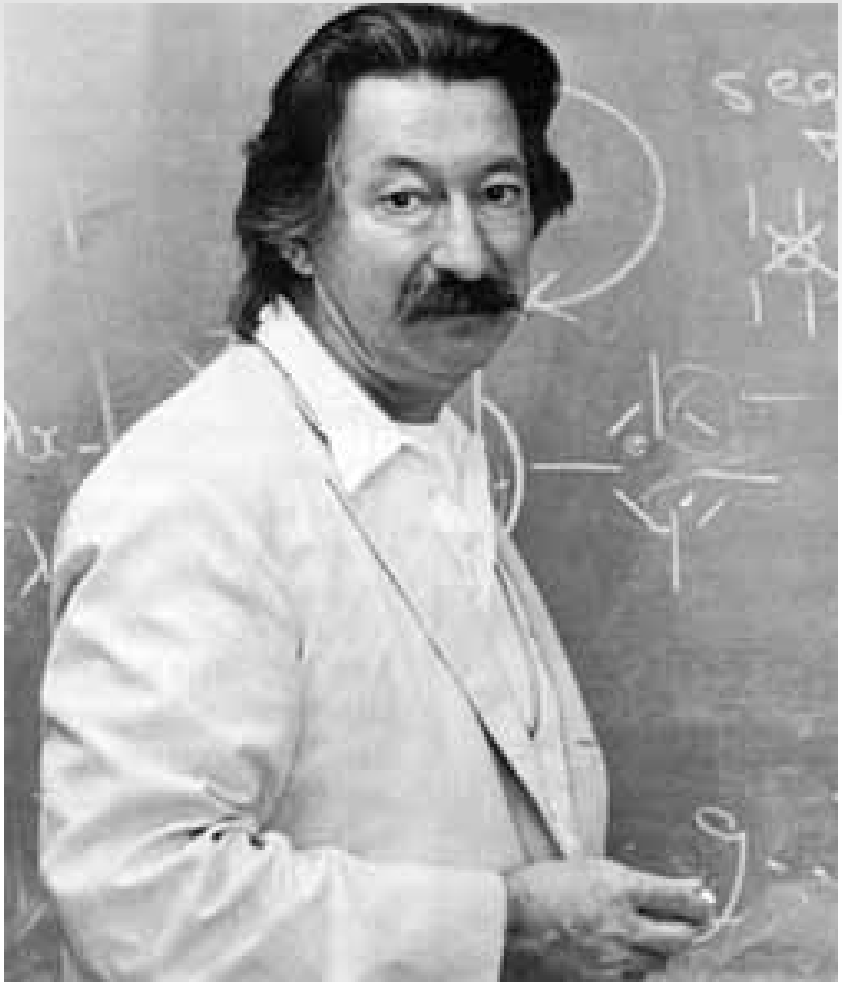


+

= www

Hypertext + Internet = World Wide Web

… und ihre heutige Form



Das World Wide Web ist ein großer Misthaufen mit ein Paar Perlen drin.

-- Joseph Weizenbaum

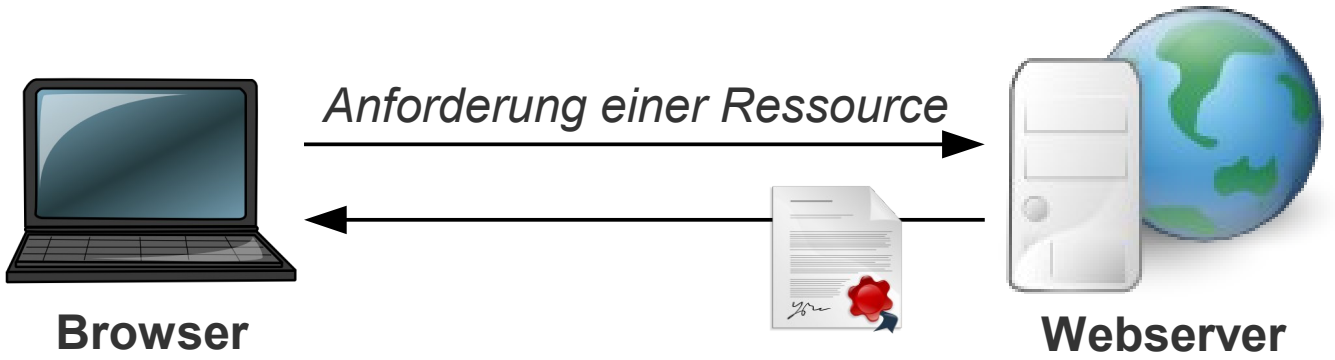
Basistechniken des Webs

Ursprüngliche Web-Technologien

- Auszeichnung von Dokumenten mit HTML
- Adressierung der Dokumente durch URLs
- Anforderung und Übertragung via HTTP

Nachträglich ergänzte Technologien

- Trennung von Inhalt und Darstellung mit CSS
- Clientseitige Codeausführung mit JavaScript



Syntax von URLs

Allgemeiner Aufbau:

```
protokoll://host:port/datei?parameter#anker
```

Beispiele:

```
http://info.cern.ch/hypertext/WWW/TheProject.html

https://www.dhbw-karlsruhe.de/index.php

http://localhost:8080/parkbank/index.jsp

http://www.google.de?q=URL%20Encoding

http://de.wikipedia.org/wiki/URL#Aufbau

http://localhost/quassel/do.jsp?action=search&asc=1
```

Hypertext Transfer Protocol

Zustandsloses Protokoll zur Übertragung von Dokumenten

Somit keine Erkennung von wiederkehrenden Clients möglich

Basiert auf dem Austausch einfacher Textnachrichten in Form von strikten **Anfrage/Antwort**-Zyklen

Anfrage an den Server

- Beginnt die Kommunikation zwischen Client und Server
- Fordert den Server auf, eine definierte Aktion auszuführen
- Aufforderung wird in Form eines sog. **Verbs** übertragen

Antwort des Servers

- Beendet die Kommunikation zwischen Client und Server
- Beinhaltet immer einen nummerischen **Status-/Fehlercode**

Beispiel: HTTP-Konversation

Client

```
GET /index.html HTTP/1.1
Host: dhbw-karlsruhe.de
Content-Type: text/html

ENDE DER ÜBERTRAGUNG
```

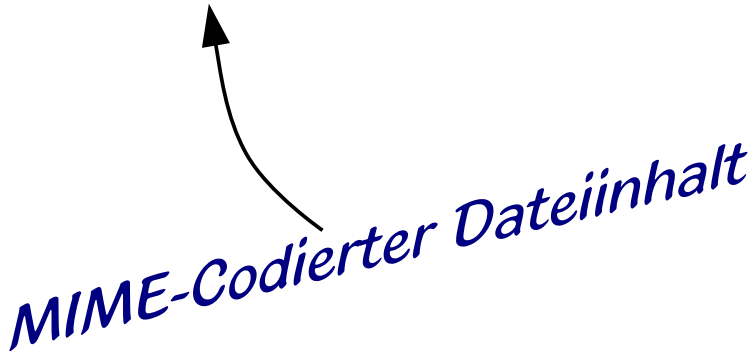
Server

```
HTTP/1.1 200 OK
Content-Length: 200
Content-Type: text/html
Connection: close

<html>
  <head> ... </head>
  <body>
    ...
  </body>
</html>
ENDE DER ÜBERTRAGUNG
```

Beispiel: HTTP-Konversation

Client

```
PUT /image.jpg HTTP/1.1
Content-Type: image/jpeg
Content-Length: 50

XHYJyhJHSDhjh//yhxjhjA/(/
YXJdssSSDREUISHJAnSDNMSnA
HFDJAHAJHFD/KJAS//jhsSDha
AhsdhjAehvfmnauAZUsdhjdNA
SHurnfnmsnma//SdsjksdkHFQ
ENDE DER ÜBERTRAGUNG
```



Server

```
HTTP/1.1 200 OK
Connection: close

<html>
  <head> ... </head>
  ...
</html>
ENDE DER ÜBERTRAGUNG
```

Beispiel: HTTP-Konversation

Client

```
DELETE /image.jpg HTTP/1.1

ENDE DER ÜBERTRAGUNG
```

```
GET /faq.html HTTP/1.1
Host: dhbw-karlsruhe.de
Content-Type: text/html

ENDE DER ÜBERTRAGUNG
```

Server

```
HTTP/1.1 403 Forbidden
Connection: close

<html> FEHLER 403 ... </html>
ENDE DER ÜBERTRAGUNG
```

```
HTTP/1.1 404 Not Found
Connection: close

<html> FEHLER 404 ... </html>
ENDE DER ÜBERTRAGUNG
```

Hypertext Markup Language

Auszeichnungssprache für **Hypertext**-Dokumente

Rein logische Auszeichnung, so dass der Textinhalt von seiner Darstellung getrennt wird

Beruht auf der Syntax des SGML-Standards

- Jedoch einfachere und einheitlichere Syntax als SGML
- Sowie wesentlich weniger Funktionen als SGML

Syntax aller HTML-Elemente:

```
<Elementname name="wert">
    ...
</Elementname>
```

Beispiel: Text ohne Markups

Threads in Java

Je nach dem, wie lange Sie schon mit Computern arbeiten, oder wie sehr Sie sich für historische Computer interessieren, erinnern Sie sich vielleicht noch an die Zeiten, als ein Computer nur ein Programm gleichzeitig ausführen konnte. Obwohl die Techniken zur nebenläufigen Ausführung mehrerer Programme bereits längst bekannt und in verschiedenen Betriebssystemen umgesetzt waren, war dies bis in die 1990er hinein der Normalfall für Personal Computer.

Erst als UNIX auch auf die Kleincomputer kam und als Hersteller wie Microsoft und IBM begannen, neuartige Betriebssysteme wie Windows und OS/2 zu entwickeln, kam das Ende der bis dahin weit verbreiteten Single Tasking Systeme wie MS-DOS oder dem damaligen Mac OS.

Beispiel: Text mit Markups

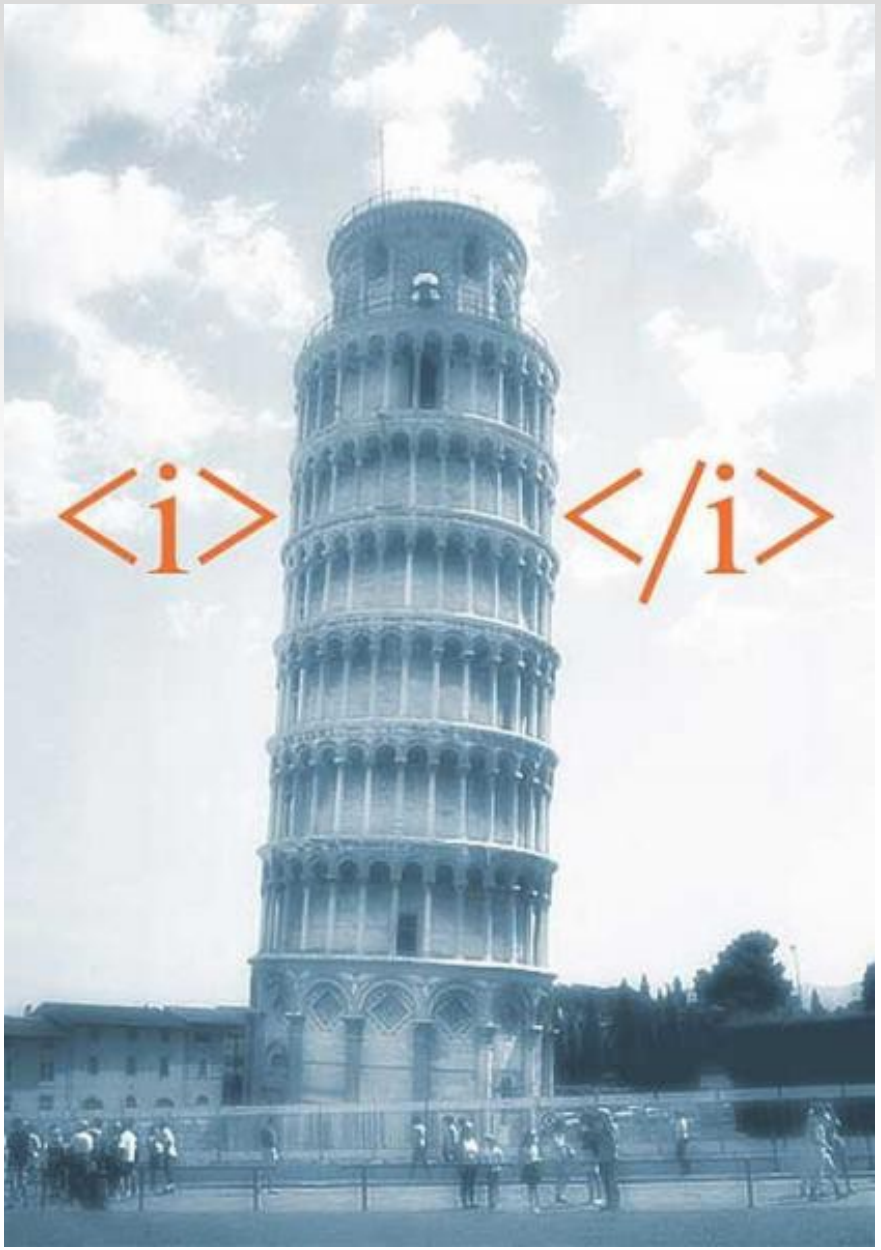
<h1>Threads in Java</h1>

Je nach dem, wie lange Sie schon mit Computern arbeiten, oder wie sehr Sie sich für historische Computer interessieren, erinnern Si[illegible] vielleicht noch an die Zeiten, als ein Computer nur ein Progra[illegible] konnte. Obwohl die Techniken zur [illegible] bereits lä[illegible]emen umgesetz[illegible]malfall für Pe[illegible]

Erst als [illegible] und als Hersteller wie Microsoft [illegible]annen, neuartige Betriebssysteme wie Windows und OS/2 zu entwickeln, kam das Ende der bis dahin weit verbreiteten Single Tasking Systeme wie MS-DOS oder dem damaligen Mac OS.

Der Text wird mit Auszeichnungselementen versehen, wobei die ausgezeichneten Passagen von den Elementen eingeklammert werden

Kursivschrift

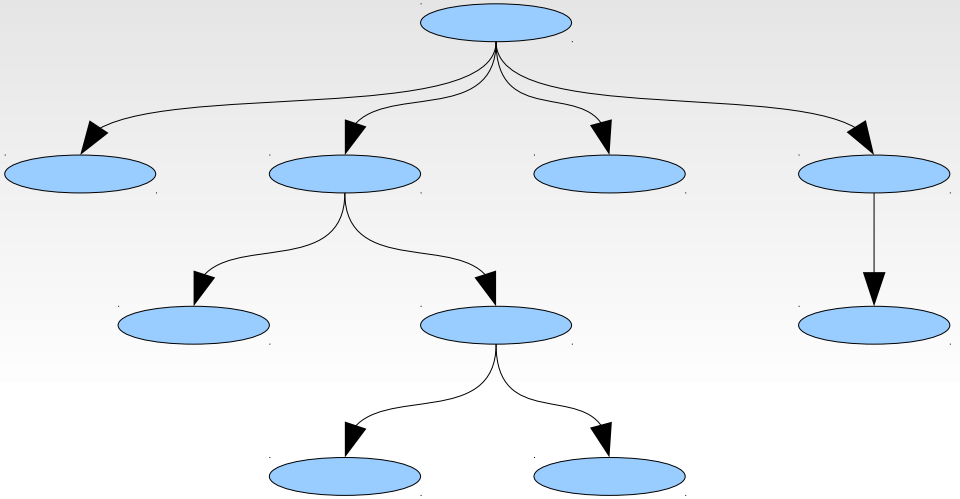


Darstellung der Dokumente

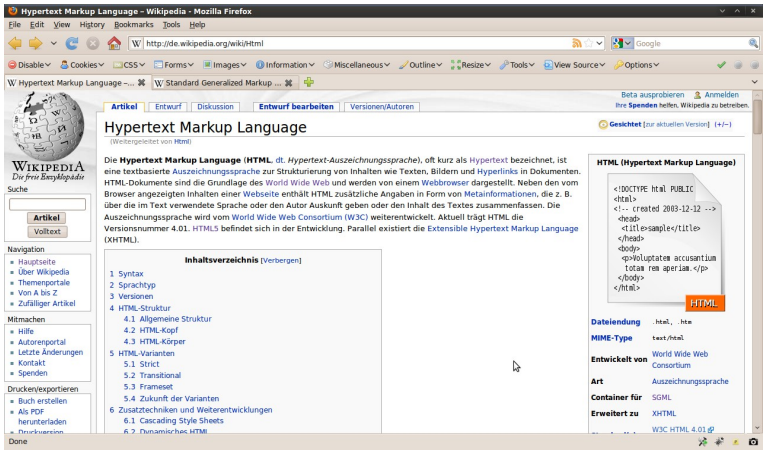
1. Der Anwender gibt im Browser eine Adresse ein oder klickt auf einen Link.

2. Der Browser besorgt das angefragte Dokument und alle eingebundenen Ressourcen (Bilder, Sounds, …).

3. Der Browser parst das empfangene Dokument und erstellt den DOM-Baum.

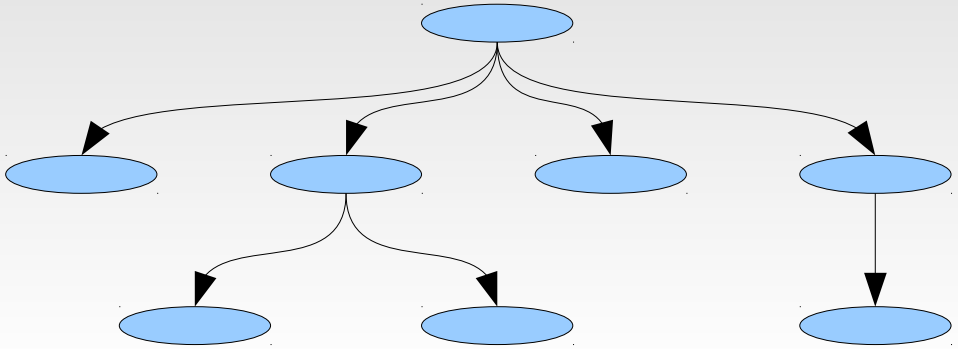


4. Anschließend rendert der Browser das Dokument und zeigt es an.

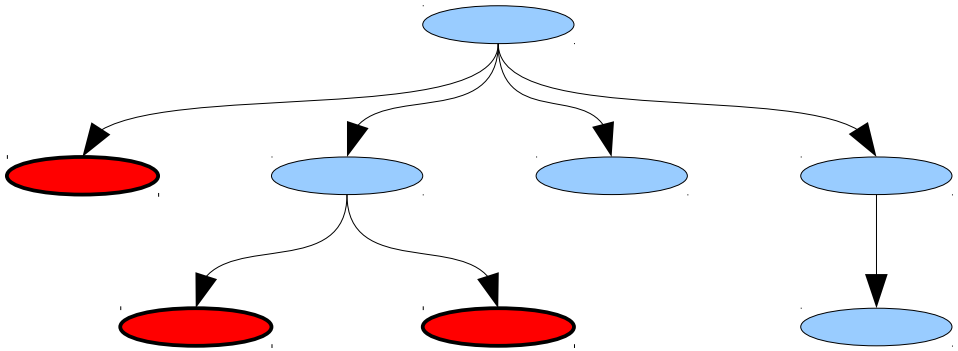


Funktionsweise von CSS

1. Ein HTML-Dokument definiert Inhalt und Struktur der anzuzeigenden Seite.



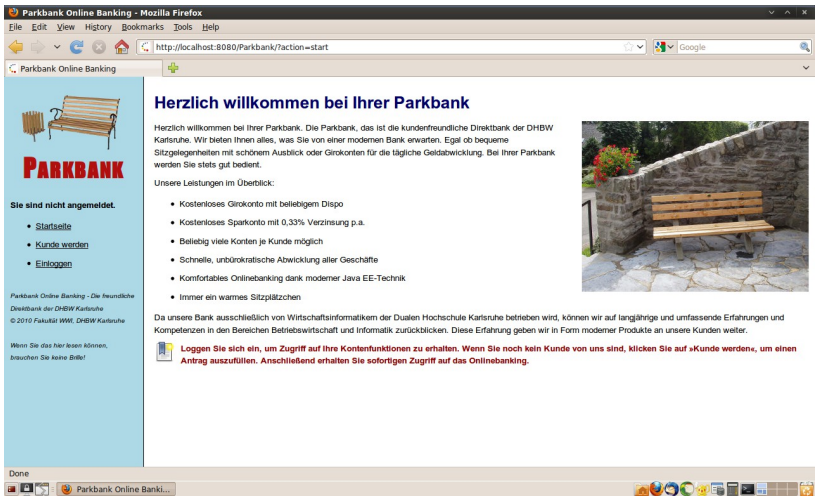
2. Mit speziellen CSS-Selektoren werden Teile der Dokumentstruktur ausgewählt.



3. Anschließend wird die Darstellung der ausgewählten Elemente notiert. Dabei erbt jedes Element automatisch die Darstellung seiner übergeordneten Elternelemente.

```
color: red;
font-weight: bold;
border-width: 1px;
```

4. Im Browser werden die Elemente dann gemäß ihren CSS-Eigenschaften angezeigt.



Beispiel: CSS-Eigenschaften

```
* {
   font-family: serif;
}

h1 {
   color: darkblue;
   font-size: 3ex;
   font-weight: bold;
}

p {
   font-size: 1.5ex;
}

p.note {
   border-width: 1px;
   border-style: solid;
   background-color: #FFFF99;
}
```

Threads in Java

Je nach dem, wie lange Sie schon mit Computern arbeiten, oder wie sehr Sie sich für historische Computer interessieren, erinnern Sie sich vielleicht noch an die Zeiten, als ein Computer nur ein Programm gleichzeitig ausführen konnte. Obwohl die Techniken zur nebenläufigen Ausführung mehrerer Programme bereits längst bekannt und in verschiedenen Betriebssystemen umgesetzt waren, war dies bis in die 1990er hinein der Normalfall für Personal Computer.

Wussten Sie, dass die beiden Betriebssysteme Windows und OS/2 ursprünglich von Microsoft und IBM gemeinsam entwickelt wurden? Beide Systeme waren kompatibel zueinander, wobei es OS/2 heute nicht mehr gibt.

Erst als UNIX auch auf die Kleincomputer kam und als Hersteller wie Microsoft und IBM begannen, neuartige Betriebssysteme wie Windows und OS/2 zu entwickeln, kam das Ende der bis dahin weit verbreiteten Single Tasking Systeme wie MS-DOS oder dem damaligen Mac OS.

JavaScript: Code im Browser

Die Sprache JavaScript

- Objekt-orientierte, dynamische Skriptsprache von Netscape
- Nutzt (in der Regel) den Browser als Laufzeitumgebung
- Ermöglicht es, dynamisch erzeugte Inhalte anzuzeigen
- Hierfür Zugriff auf das **Document Object Model** möglich

Das Document Object Model

- Baumartige Datenstruktur im Hauptspeicher des Browsers
- Beinhaltet alle Elemente einer angezeigten HTML-Seite
- Dient als Grundlage, auf der eine Seite gerendert wird
- Nachträgliche Änderungen werden sofort angezeigt

Webanwendungen mit AJAX

Asynchronous JavaScript and XML

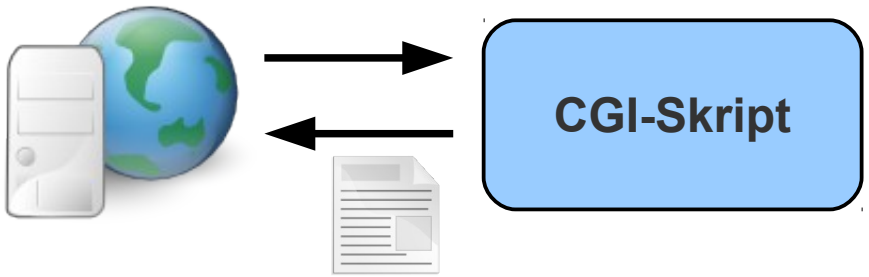
Traditionelle Webanwendungen schicken nach jedem Mausklick immer eine komplett neue Seite an den Browser. AJAX-Webanwendungen hingegen nutzen JavaScript und DOM, um vom Server nur den zu aktualisierenden Teil einer bereits angezeigten Seite anzufordern.

Somit sollen Webanwendungen mehr wie Desktopprogramme aussehen.

Anwendungen ohne AJAX

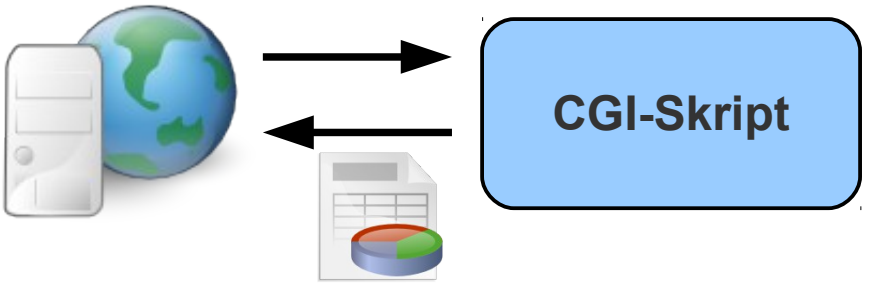
1. Der Anwender fordert die Startseite einer Webanwendung vom Server an.

2. Der Webserver führt z.B. ein CGI-Skript aus, um die Seite zu generieren.



3. Durch Anklicken eines Links fordert der Anwender eine weitere Aktion vom Server an.

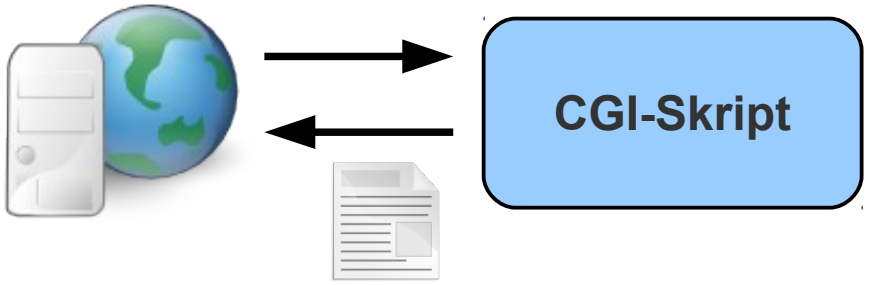
4. Serverseitig wird eine neue Seite generiert und an den Browser geschickt.



Anwendungen mit AJAX

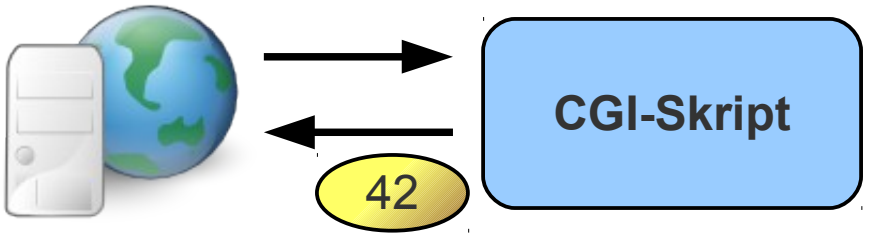
1. Der Anwender fordert die Startseite einer Webanwendung vom Server an.

2. Der Webserver führt z.B. ein CGI-Skript aus, um die Seite zu generieren.



3. Durch Anklicken eines Links fordert der Anwender eine weitere Aktion vom Server an.

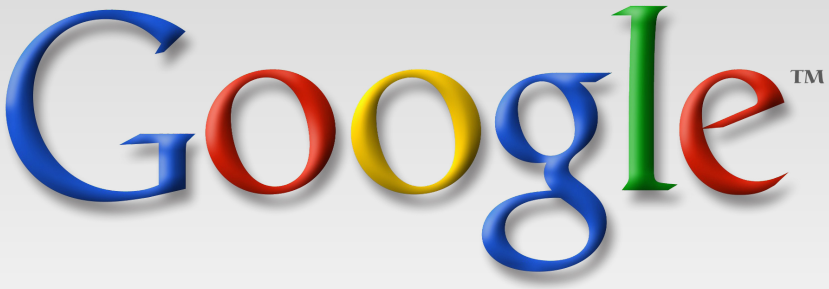
4. Der Server führt die Aktion aus und schickt nur das Rechenergebnis an den Client. Dort wird mit Hilfe von JavaScript das Ergebnis ohne Neuaufbau der Seite angezeigt.



Webanwendungen mit AJAX


Webentwicklung mit der
Java Enterprise Edition

Bekannte Webanwendungen



Dynamische Webseiten

Webseiten als Dokumente im World Wide Web

- Ursprünglich nur statische Webseiten und Ressourcen
- Angefragte Dokumente lagen als Dateien auf dem Server
- HTTP war als reiner Auslieferungsdienst konzipiert, eine dynamische Erzeugung der Inhalte war nicht vorgesehen

Unterschied zu modernen Webanwendungen

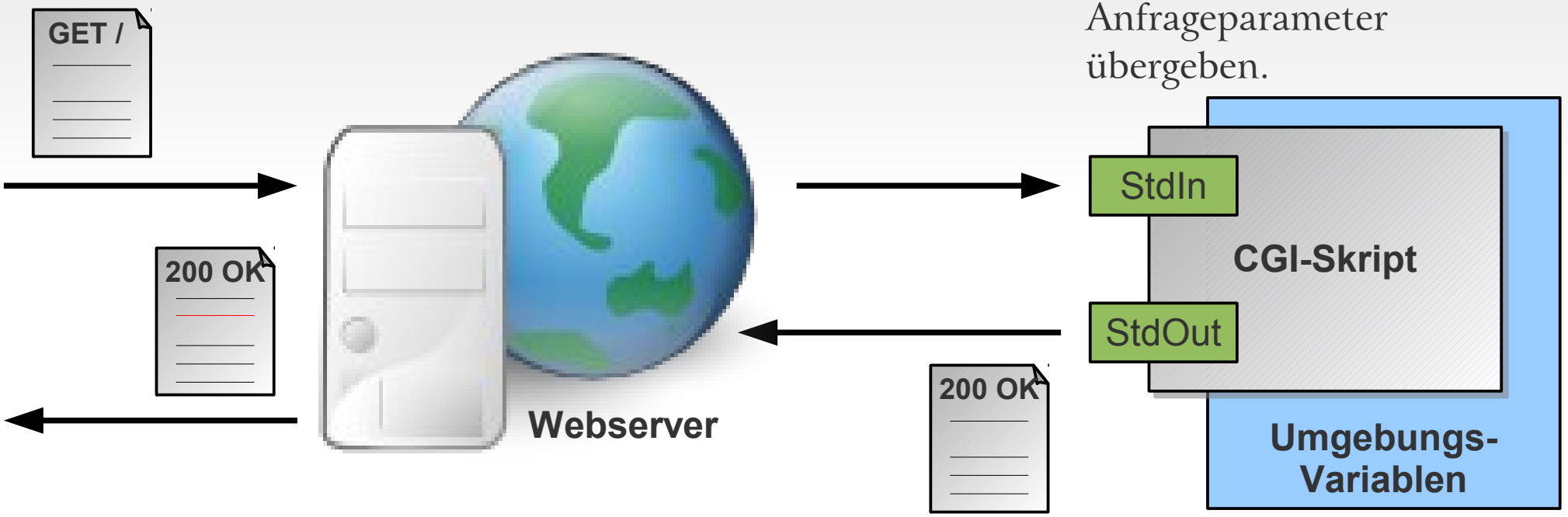
- Browser als Laufzeitumgebung für komplexe Anwendungen
- Der angezeigte HTML-Code wird erst zur Laufzeit erzeugt
- Serverseitige und clientseitige Erzeugung möglich
- Frühe Techniken: CGI oder HTML-Formulare

Klassische Umsetzung

1. Der Webserver empfängt eine HTTP-Anfrage.

2. Daraufhin startet der Server ein CGI-Skript und übergibt ihm die komplette Anfrage.

Über Umgebungsvariablen werden dem Skript Informationen über den Client und die Anfrageparameter übergeben.



5. Der Server fügt der generierten Antwort noch weitere Header Fields hinzu. Anschließend schickt er die Antwort zurück an den Client.

4. Das Skript erzeugt eine HTTP-Antwort und schreibt diese in sein Standard Output.

Aktive Serverseiten

1. Der Webserver empfängt eine HTTP-Anfrage.

2. Da sich die Anfrage auf ein normales HTML-Dokument bezieht, sucht der Server das Dokument im Dateisystem.

3. Das Dokument enthält sowohl HTML aus auch Programmcode, so dass es erst einem Interpreter übergeben werden muss.

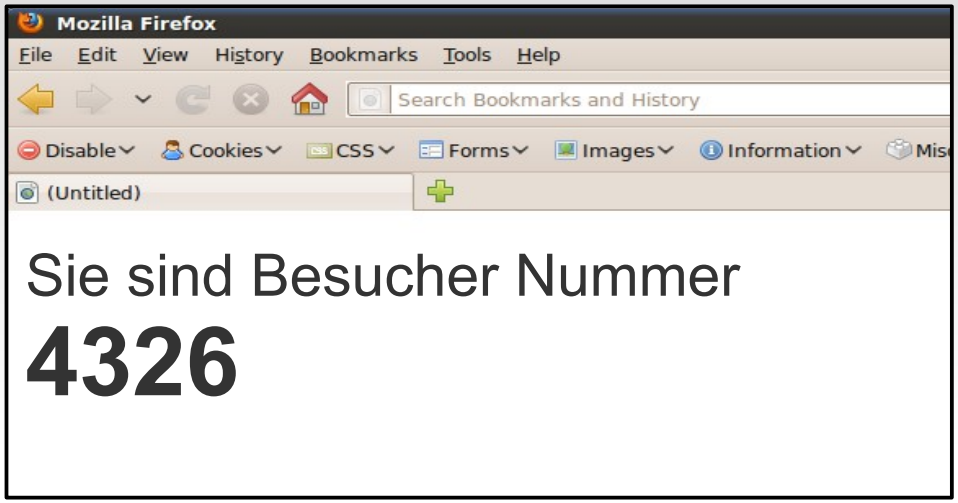

```
<html>
  <head></head>
  <body>
    <h1>PHP</h1>
    <%
    function signin() {
      $r = $_REQUEST;
      $u = $r["USER"];
      $p = $r["PASS"];
    }
    %>
  </body>
</html>
```

5. Der Webserver erzeugt aus dem Ergebnis des Interpreters eine vollständige HTTP-Antwort und schickt diese an den Client.

4. Der Interpreter führt den im Dokument eingebundenen Programmcode aus und gibt das Ergebnis an den Server zurück.

Beispiel: Frühe Webanwendungen

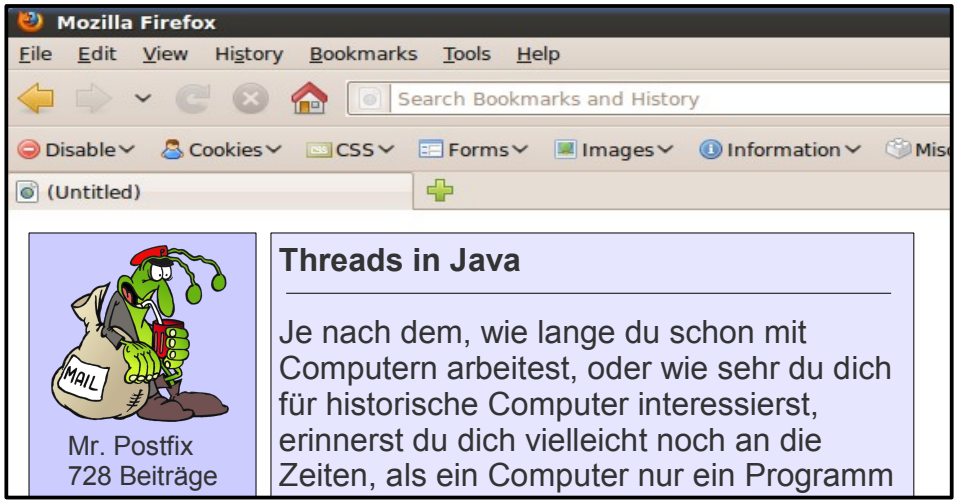
Besucherzähler



Gästebuchskripts

Feedback-Formulare

Einfache Webforen



Bewertung der Techniken

Aufruf externer CGI-Skripte

- Pragmatische Lösung, einfach umzusetzen
- Fast jede Programmiersprache kann verwendet werden
- Ein Prozess je Anfrage: Hoher Ressourcenverbrauch
- Sehr viel Low Level-Code muss geschrieben werden

Interpretation aktiver Serverseiten

- Viel einfacher zu programmieren als CGI-Skripte
- Langsame Ausführung, weil Seiten interpretiert werden
- Keine Trennung von Ablauflogik und Darstellung

Webanwendungen in Java

Entwicklung als in sich geschlossene **Komponenten**

Spezielle **Laufzeitumgebung** (Java Enterprise Edition)

Webanwendungen laufen im sog. Webcontainer

- Stellt einen in Java programmierten Webserver dar
- Laufzeitumgebung für Servlets und Java Server Pages
- Technische Grundlage für weiterreichende Webframeworks

Bekannte Open Source-Webcontainer

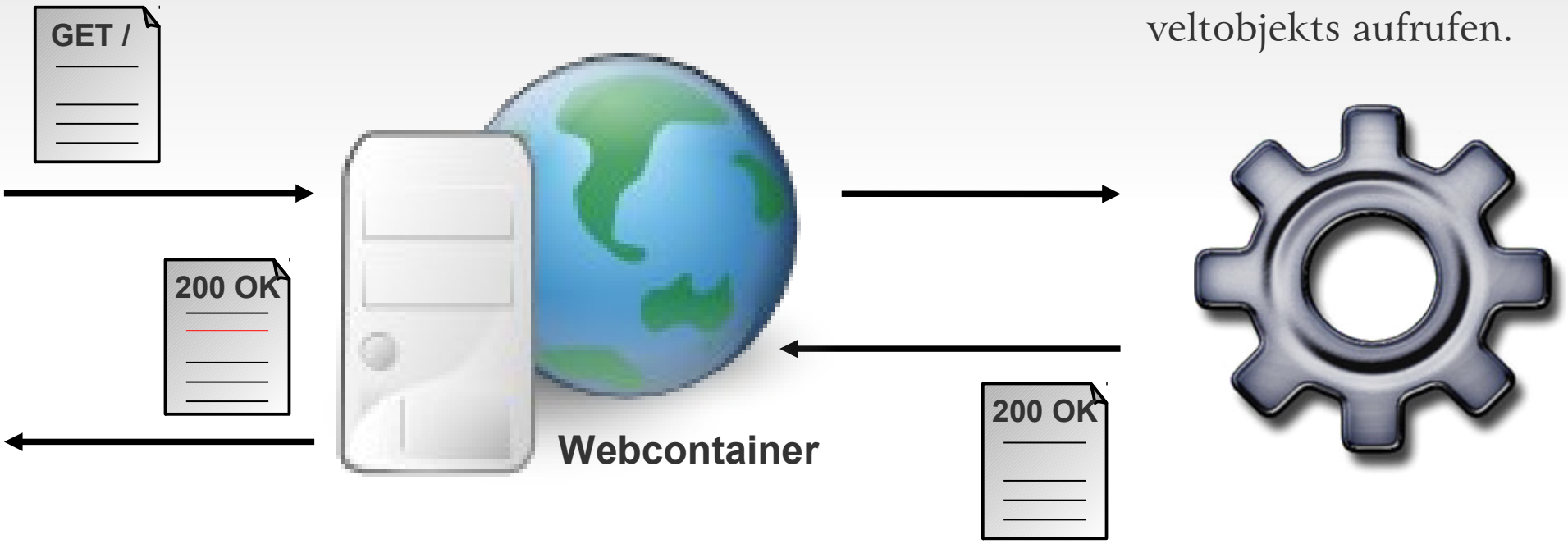
- Apache Tomcat
- Mort Bay Consulting's Jetty

Javabasierte Webanwendungen

1. Der Webcontainer empfängt eine HTTP-Anfrage.

2. Zu jeder anfragbaren URL existiert eine Klasse (Servlet), welche die Anfrage bearbeiten kann.

Um die Anfrage zu bearbeiten, muss der Container nur eine spezielle Methode des entsprechenden Serveltobjekts aufrufen.



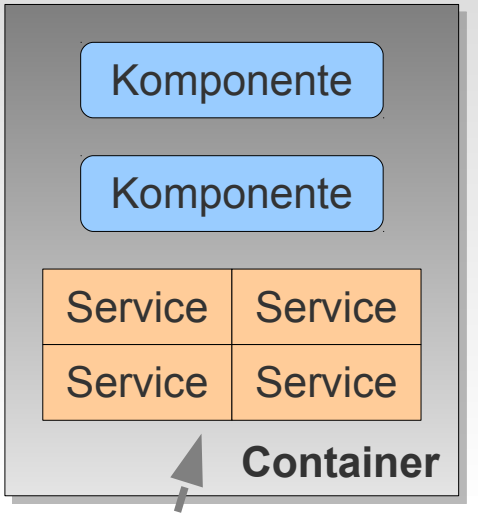
5. Der Server wandelt die generierte Antwort in eine HTTP-Antwort um. Anschließend schickt er die Antwort zurück an den Client.

4. Innerhalb der Methode müssen die zurückzuschickenden Daten erzeugt werden.

Container und Komponenten

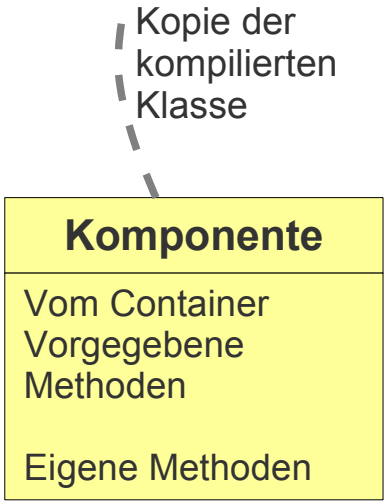
Laufzeitumgebung / Container

- Ist ein fertiges Programm ohne Funktion
- Wird durch Softwarekomponenten erweitert
- Erzeugt/Zerstört die Komponentenobjekte
- Bietet den Komponenten verschiedene Services



Softwarekomponenten

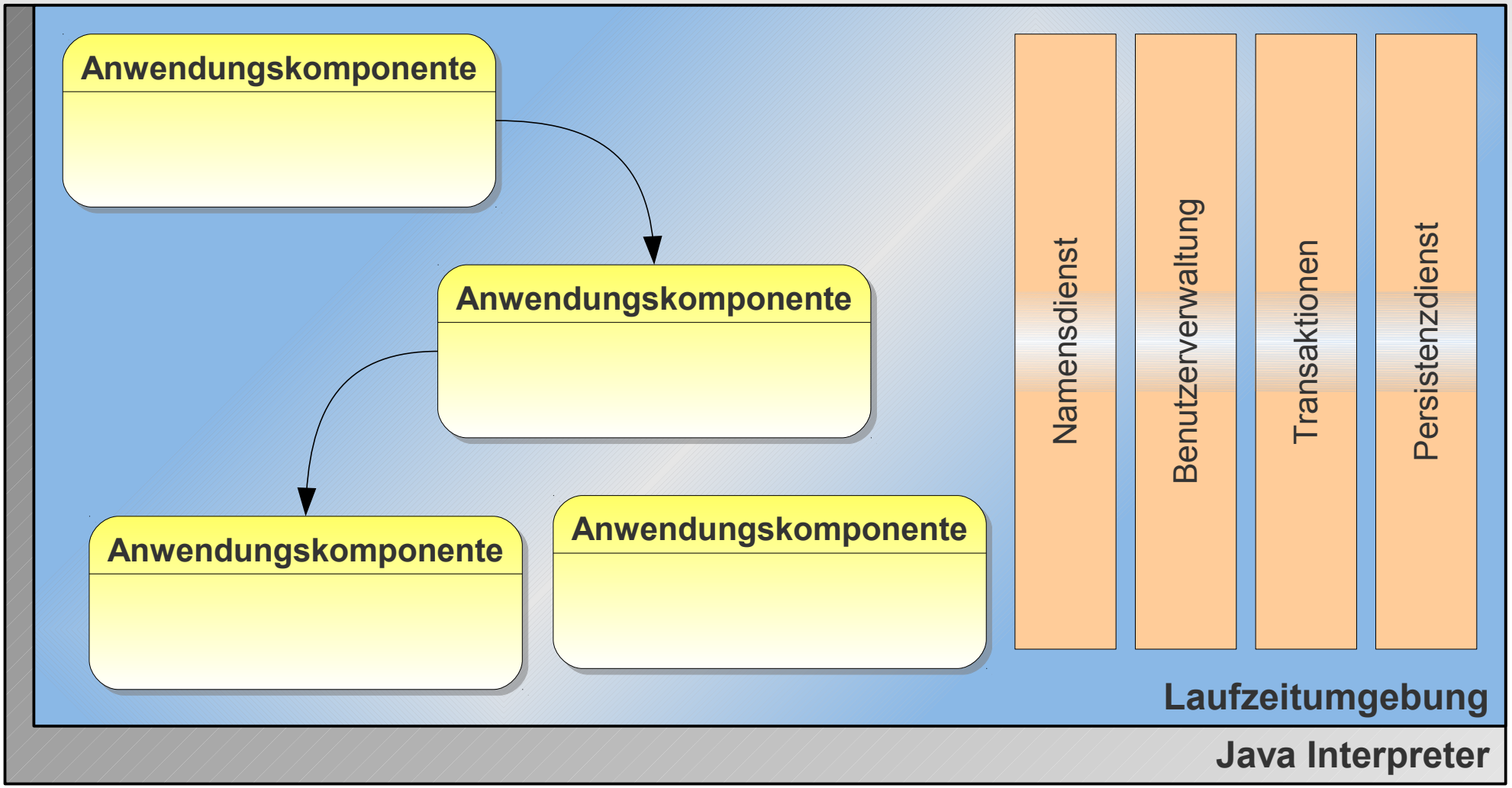
- Kapseln einzelne Funktionen einer Anwendung
- Werden als gewöhnliche Klassen programmiert
- Methoden vom Container vorgeschrieben
- Aufruf der Methoden ebenfalls vom Container



Beispiel: Laufzeitumgebung

Anwender startet die Laufzeitumgebung

```
dennis@localhost:~$ jboss/bin/run.sh
```



Bestandteile von Java EE

Webprogrammierung
Servlets
Java Server Pages
Java Server Faces

4. Semester

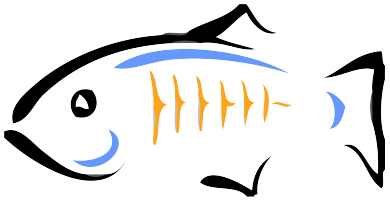
Entfernter Prozeduraufruf
JAX-RPC
JAX-WS

Entfernte Methodenaufrufe
Enterprise Java Beans

Nachrichtenaustausch
Java Messaging Services

Persistenzdienst
Java Database Connection
Java Persistence API
Java Transaction API

Verzeichnisdienst
Java Naming and Directory Interface

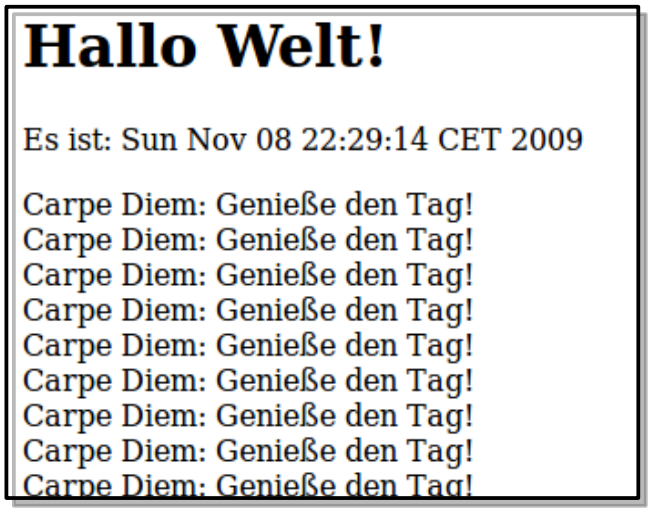


Beispiel: HTTP-Servlet

```
import java.io.*;
import java.util.Date;
import javax.servlet.http.*;

public class SimpleHttpServlet extends HttpServlet {
    public void doGet
    (HttpServletRequest request, HttpServletResponse response)
    throws IOException, ServletException {
        // Datenstrom für Ausgabe zum Client
        PrintWriter toClient = response.getWriter();

        // HTML-Code erzeugen und ausgeben
        toClient.println("<html>");
        toClient.println("<head><title>Test</title></head>");
        toClient.println("<body>");
        toClient.println("<h1>Hallo Welt!</h1>");
        toClient.println("Heute ist: " + new Date());
        toClient.println("</body>");
        toClient.println("</html>");
    }
}
```

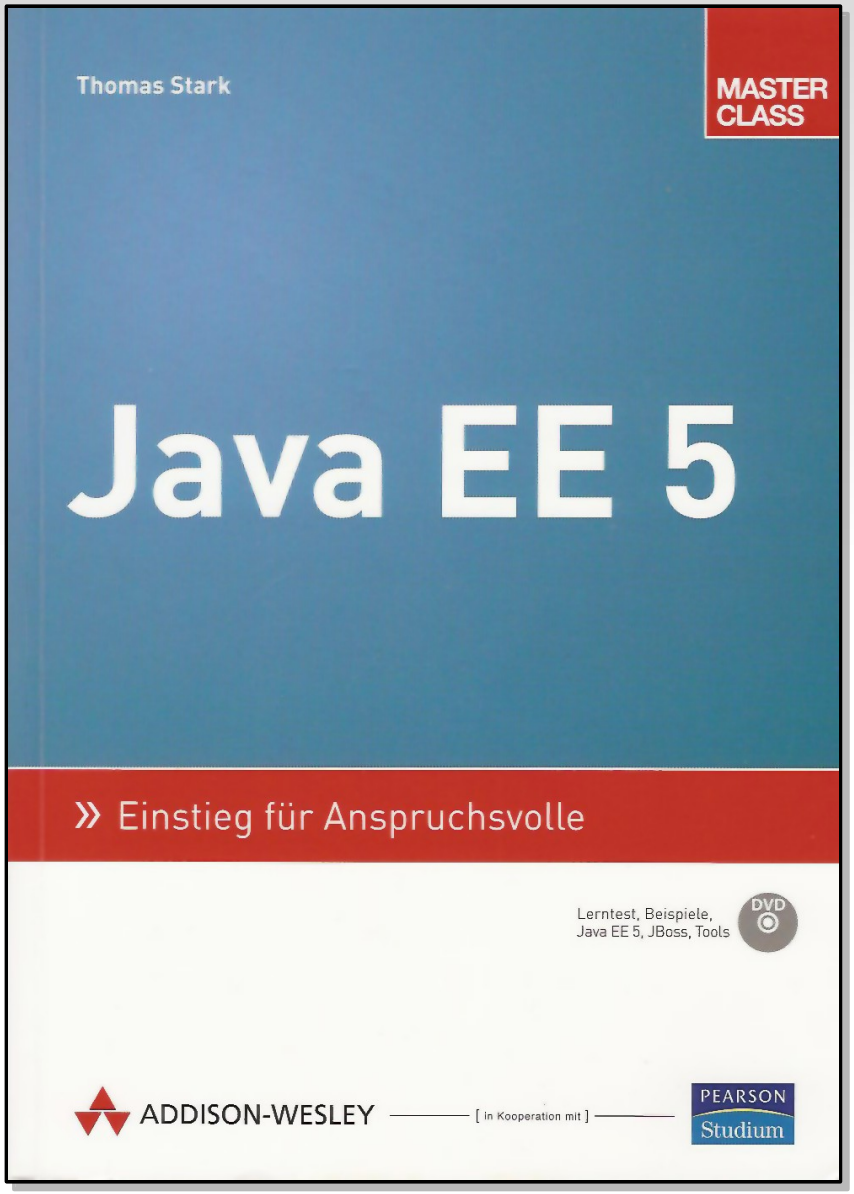
Beispiel: Java Server Page

```
<%@page contentType="text/html" pageEncoding="UTF-8"%>
<!DOCTYPE html>

<html>
    <head></head>
    <body>
        <h1>Hallo Welt!</h1>
        <p> Es ist: <%= new java.util.Date().toString() %>
</p>

        <% for (int i = 0; i < 10; i++) { %>
            Carpe Diem: Genieße den Tag! <br/>
        <% } %>
    </body>
</html>
```



So programmiert man doch keine HTML-Webanwendungen!

Buchempfehlung



Java EE 5
Einstieg für Anspruchsvolle

Thomas Stark

ISBN 978-3-8273-2648-5

1) Java Server Pages
2) Servlets
3) JSP Tag-Bibliotheken
4) Das Struts-Webframework
5) Java Server Faces
6) Java Naming and Directory Interface
7) Enterprise Java Beans
8) Java Message Service
9) Java Persistence API
10) Extensible Markup Language (XML)
11) XSL, Xpath & Co.
12) Webservices